Accelerating Cryptosystems on Hardware Platforms

by

Wei Wang

A Dissertation

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Doctor of Philosophy

in

Electrical and Computer Engineering

January 2014

APPROVED:

Professor Xinming Huang Thesis Advisor ECE Department Worcester Polytechnic Institute	Professor Berk Sunar Thesis Committee ECE Department Worcester Polytechnic Institute
Professor Wenjing Lou Thesis Committee CS Department Virginia Tech	Professor Yehia Massoud Head of Department ECE Department Worcester Polytechnic Institute

Abstract

In the past decade, one of the major breakthroughs in computer science theory is the first construction of fully homomorphic encryption (FHE) scheme introduced by Gentry. Using a FHE one may perform an arbitrary numbers of computations directly on the encrypted data without revealing of the secret key. Thus an untrusted party, such as a remotely hosted server, may perform computations on the data without compromising privacy. Therefore, a practical FHE provides an invaluable security application for emerging technologies such as cloud computing and cloud-based storage. However, FHE is far from real life deployment due to serious efficiency impediments. This dissertation focuses on accelerating the existing FHE schemes using GPU and hardware design for the existing schemes to make the existing FHE schemes more efficient and practical towards real-life applications. The integer-FFT multiplication algorithm is adopted for the implementation of Gentry-Halevi's FHE scheme. As the Moore law continues driving the computer technology, the key size of the Rivest–Shamir–Adelman (RSA) encryption is necessary to be upgraded to 2048, 4096 or even 8192 bits to provide higher level security. In this dissertation, the FFT multiplication is employed for the large-size RSA hardware design instead of using the traditional interleaved Montgomery multiplication to show the feasibility of the FFT multiplication for large-size RSA design. The main technical contributions of this dissertation are summarized as following:

At first, GPU is used to accelerate the Gentry-Halevi FHE scheme. Gentry-Halevi FHE scheme is the first software implementation of the FHE scheme on a general-purpose computer. Although Gentry-Halevi's FHE scheme employs a numbers of optimizations, it is still considerably slow. In the Gentry-Halevi implementation,

the most computationally intensive arithmetic operation is modular multiplication. In this research, the million-bit modular multiplication is computed in two steps. For large number multiplication, Strassen's FFT based algorithm is employed and accelerated on a graphics processing unit (GPU) through its massive parallelism. Subsequently, Barrett modular reduction algorithm is applied to implement modular reduction. As an experimental study, we implement the Gentry-Halevi primitives for the small setting with a dimension of 2048 on NVIDIA C2050 GPU. The experimental results show the speedup factors of 7.68, 7.4 and 6.59 for encryption, decryption and recryption respectively, when compared with the existing CPU implementation.

Second, a hardware multiplier is designed for the Gentry-Halevi FHE scheme. A power efficient, high-speed 768K-bit multiplier based on FFT multiplication is designed for fully homomorphic encryption operations. A memory-based, in-place architecture is presented for the FFT processor that performs 64K-point finite-field FFT operations using a radix-16 computing unit and 16 dual-port SRAMs. By adopting a special prime as the base of the finite field, the radix-16 calculations are simplified to require only additions and shift operations. A two-stage carry-look-ahead scheme is employed to resolve carries and obtain the multiplication result. The multiplier design is validated by comparing its results with the GNU Multiple Precision (GMP) arithmetic library. The proposed design has been synthesized using 90nm process technology with an estimated die area of 45.3 mm^2. At 200MHz, the large number multiplier offers roughly twice the performance of a previous implementation on an NVIDIA C2050 GPU and is 29 times faster than the Xeon X5650 CPU, with the power consumption of a modest 0.97W.

Thirdly, GPU is used to accelerate the leveled FHE scheme. In leveled FHE scheme, large-number matrix-vector multiplication is a crucial part of the encryption.

In this research, the Chinese Remainder Theorem (CRT) is employed to reduce the computational complexity of the large-number element-by-element modular multiplication. As a result, the large-number matrix-vector multiplication is divided into three steps: decomposition, vector operation and reconstruction. The CRT-based method is compared with Number Theory Library (NTL), showing the proposed method is about 7.8 times faster when executing on CPU. Therefore GPU acceleration is employed to speed up the vector operations accounting for 99.6% of the total computation time. In the GPU implementation, the GPU computation and data transfer process between GPU and CPU are overlapped. Experiment results show that the GPU implementation of the CRT-based method is 35.2 times faster than the same method implemented on CPU and is about 274 times faster than the NTL library on CPU.

Finally, we explore the feasibility of using FFT multiplication for the large-size RSA cryptosystem. A new modular multiplication combining the Strassen multiplication algorithm and Montgomery reduction are designed and an associated RSA modular exponentiation algorithm is present. The modular multiplication architecture is different from the interleaved version of Montgomery multiplication traditionally used in RSA design. By selecting different bases of 16 or 24 bits, it can perform 8,192-bit or 12,288-bit modular multiplication. A new RSA modular exponentiation algorithm using FFT multiplication is proposed to reduce one third of the calculation time of the large-number multiplication in modular multiplication. The design was implemented on the Altera's Stratix-V FPGA and 90-nm application-specified integrated circuit technologies. It performs one 8K-bit modular multiplication in 6.34 μs and one modular exponentiation in 0.104 s when operating at 320 MHz.

Acknowledgements

First of all, I would like to sincerely thank for my advisor Professor Xinming Huang for his guidance and support through all my Ph.D. studies and research at the Worcester Polytechnic Institute. I am very grateful for his patience, inspiration and support that have helped me move forward in my Phd career.

I would like to thank Professor Berk Sunar and Professor Wenjing Lou for their valuable comments as my dissertation committee.

I am grateful to my WPI colleagues Dr. Kai Zhang, Dr. Yanjie Peng, Dr. Yin Hu, Kavin Yang, Zhilu chen, Jin Zhao, Sichao Zhu, Yun Zhou, Yuteng Zhou for their friendship and support.

I would also like to thank all the professors and staff in ECE department for their help and support during my study in WPI.

This dissertation is dedicated to my parents for their love, support and encouragement during all my life.

Especially I would like to thank Prof. Berk Sunar and Dr. Yin Hu for their great job in the joint project. Prof. Sunar gives me the general research direction for my dissertation and the great idea and support in the joint project. Dr. Yin Hu puts all his great ideas and efforts in the project when working together. The project cannot be done smoothly and perfect without him. I would also like to thank Niall Emmart in the UMass, who gives me many good suggestions and advices in the research.

Contents

List of Tables

List of Figures

List of Algorithms

List of Abbreviations

AES Advanced encryption standard

ASIC Application specific integrated circuits

CMOS Complementary metal oxide semiconductor

CPU Central processing unit

CRT Chinese remainder theorem

CUDA Compute unified device architecture

FFT Fast Fourier transform

FHE Fully homomorphic encryption

GMP GNU multiple precision arithmetic

GPGPU General-purpose computing on graphics processing units

GPU Graphics processing unit

IFFT Inverse fast Fourier transform

NTL Number theory library

OMF Operation maximum frequency

RAM Random-access memory

ROM Read-only memory

RSA Rivest Shamir Adelman

SRAM Static random access memory

SWHS Somewhat homomorphic encryption

VLSI Very large scale integration

Chapter 1

Introduction

In this chapter, we first introduce background and discuss motivations of our work in Section 1.1. The motivations and contributions of our work are summarized in Section 1.2. Finally the organization of this dissertation is presented in Section 1.3.

1.1 Background

Recently, cloud storage and computing are developing at a fast speed, which allows users outsource computations and storage on their data. In this way, users' private data can be exposed to untrusted cloud. As a result, privacy and security concerns become a big issue in cloud storage and computing industry. A good solution to this privacy and security problem is to keep all data in an encrypted form and perform computations directly on the encrypted data. Therefore, fully homomorphic encryption, which supports an arbitrary number of computations directly on the encrypted data, is invaluable for the cloud storage and computing platforms today. Besides, fully homomorphic encryption is also very useful in a number of other applications such as

electronic voting [1], private information retrieval [2] and financial applications [3].

Rivest, Adleman and Dertouzos first proposed the concept of encryption scheme that allows arbitrary operations on encrypted data without revealing the secret key in 1978 [4]. Many homomorphic encryption schemes, permit simple operations on encrypted, has been proposed in the past decades. Goldwasser and Micali encryption scheme was the first discovery of semantically secure homomorphic scheme, supporting homomorphic evaluation of a bit-wise exclusive-OR (XOR) operation [5]. Other homomorphic encryption schemes that support either adding or multiplying encrypted ciphertexts were introduced later. The scheme support multiplicative homomorphic evaluation including RSA [6] and El Gamal encryption scheme [7]. On the otherwise, the additive homomorphic encryption scheme includes the Paillier encryption scheme [8], Damgard-Jurik encryption scheme [9], the lattice-based encryption schemes [10] [11] and many others [12] [13]. All these encryption schemes can only support either additive or multiplicative homomorphich calculations, but not both. Boneh, Goh and Nissim [14] introduced the first construction of homomorphic encryption scheme that can support both operations at the same time. However, their scheme can support arbitrary additions but only a single multiplication. The major breakthrough work cames with the first plausible construction of fully homomorphic encryption based on lattice by Gentry in 2009 [15], which can support an arbitrary numbers of additions and multiplications on encrypted data.

The first step in Gentry's FHE scheme is to construct a *Somewhat Homomorphic Encryption*(SWHS) that can only evaluate functions of limited complexity. The ciphertext in the SWHS scheme is noisy, which means it contains noise to ensure security. The amount of noise in the ciphertext grows as the homomorphic evaluations are performed until it is so large that the ciphertext cannot be correctly decrypted.

To prevent the accumulation of the noise, Gentry used the *bootstrapping* procedure to perform homomorphically decryption on the ciphertext, using an encrypted secrete key given in the public key, resulting a refreshed ciphertext with reduced noise.

Although Gentry's FHE scheme gives a good promise in theory, the efficiency of the FHE scheme is a big problem for practical applications. In the past three years, many new FHE constructions and optimizations are developed [16–23]. Specifically, Gentry and Halevi introduced the first software implementation of the lattice-based FHE scheme in [18]. Although it employs a number of impressive optimization methods to reduce the size of public key and improve the efficiency of primitives, the public key size is still very large about 17 Mega Bytes, encryption of one bit takes more than one second and recrypt primitive takes nearly half a minutes on a high-end Intel Xeon based server in the small setting case. In addition, after every bit-AND operations, a recryption process must be performed on the ciphertexts to reduce the noise in a manageable level. Therefore, this lattice-based FHE implementation is extremely inefficient for practical applications. Usually, the length of ciphertext (per bit encrypted), the keys, the encryption and decryption are used to compare the efficiency of different encryption schemes. However, for FHE schemes, the per-gate evaluation time, defined as the ratio of the time used for the homomorphically evaluating a circuit C to the time of evaluating C on plaintext, shows more importance in practical applications of FHE. It turns out the schemes following Gentry's lattice-based method [15, 17, 18, 21] have a per-gate computation time of $\Omega(\lambda^4)$ (where λ is the security parameter) [24]. In a recent development, a leveled FHE scheme is constructed by Brakerski, Gentry and VaiKuntanathan (BGV) in [23] with asymptotically linear efficiency, which means a per-gate evaluation time of $\Omega(\lambda)$. In this thesis, we take different approaches to accelerate FHE schemes using GPU and custom ASIC designs

for practical deployments of FHE.

1.2 Summary of Motivations and Contributions

As mentioned above, the FHE schemes are too inefficient for practical deployments. This research is motivated by the development of the practical deployments of Advanced Encryption Standard (AES) and RSA encryption schemes. One approach to accelerate AES or RSA encryption is using GPU as a co-processor [25] [26]. The other approach is to design an Application Specific Integrated Circuits (ASIC) which are dedicated to AES or RSA encryption/decryption operations [27] [28]. At microarchitecture level, it can be implemented as an extension of instruction set of the CPU. Today many embedded processors have AES or RSA cores included. This work is aimed to take a similar approach and to use GPU and design a specific hardware or IP blocks for accelerating the existing FHE schemes.

1.2.1 Acceleration of Gentry-Halevi's Fully Homomorphic Encryption on GPU

Motivation: The first software implementation of a FHE scheme was proposed by Gentry and Halevi [18]. Although it employs a number of optimizations to reduce the size of the public-key and to reduce the latencies of the primitives, it is still too inefficient for practical deployments. For instance, encryption of one bit takes more than a second on a high-end Intel Xeon based server, while recrypt operation takes nearly half a minute for the lowest security setting. With the introduction of the Compute Unified Device Architecture (CUDA), a number of applications such as the AES and RSA encryption are accelerated by the general purpose GPU computing

(GPGPU) platform [25] [26]. Therefore, GPU is served as our initial step for the acceleration of FHE schemes.

Contribution: In this work, we present the first GPU implementation of the Gentry-Halevi FHE algorithm [18]. More specifically, we combine Strassen's FFT based integer multiplication algorithm with Barrett's modular reduction algorithm to implement an efficient modular multiplier that supports the operands in the size of million bits. We then utilize the modular multiplier and other operation units to implement the FHE primitives: encryption, decryption and recryption. On the NVIDIA C2050, we obtain a factor of 7.4 times speedup for decryption over the CPU implementation in [18]. We also present the efficient implementations of encryption and recryption, which both are optimized to take advantage of the GPU parallelism. Our GPU implementation yields a speedup factor of 7.68 for encryption and 6.59 for recrypt when compared with the CPU implementation in [18]. This work appears in the proceeding of 2012 IEEE HPEC [29] and IEEE Transactions on Computers [30].

1.2.2 VLSI Design of a Large Number Multiplier for Fully Homomorphic Encryption

Motivation: The research in the above chapter has shown that performance can be improved greatly through the use of parallelism on a general purpose graphics processor (GPU). However, a typical GPU usually has very large power consumption around 200 to 400 watt, making it a power inefficient platform for practical deployment. The custom ASICs are usually designed for low-power high-performance applications. For instance, the specific hardware is designed to accelerate AES and RSA encryption and reduce power consumption for practical deployments [27] [28] traditionally. Inspired

by previous development of practical AES and RSA encryption, this work is aimed to take a similar approach and to design a specific hardware or IP blocks for accelerating the core computations in FHE. We try to design a hardware that is much faster than the GPU with far less power consumption. Since the most computationally intensive operations in the FHE primitives are large-number modular multiplications, our initial attempt is to tackle the design of a large-number multiplier that can handle 768K bits, in support of the 2048 dimension FHE scheme demonstrated by Gentry and Halevi.

Contribution: In this work, we attempt to use customized circuits to accelerate the multiplications for FHE. Specifically we present an efficient, high-speed design of a 768K-bit multiplier based on Strassen's algorithm including three main components: finite-field FFT, inverse FFT and resolving carries. A memory-based in-place FFT architecture is used for 64K-point finite-field FFT and IFFT. The FFT/IFFT processor uses a radix-16 computing unit and 16 dual-port SRAMs to store the input data, intermediate and final results. By adopting a special prime, the radix-16 calculation is greatly simplified to only additions and shift operations. Parallel architecture and two-stage carry-look-ahead scheme are applied to resolve carries for the multiplication result. The multiplier design is validated by comparing its results with the GNU Multiple Precision Arithmetic (GMP) library. The proposed design is synthesized using 90nm 9FLP process with the estimated die area of 45.3 mm^2. When the processor runs at 200MHz, it is about two times faster than the C2050 GPU with 448 cores running at 1.15GHz and 29 times faster than the Xeon X5650 processor running at 2.67GHz. This works appears on IEEE Transactions on VLSI Systems [31].

1.2.3 Accelerating Leveled Fully Homomorphic Encryption Using GPU

Motivation: FHE is hard to have a practical application in real life due to its serious efficiency impediments. Several different FHE schemes have been proposed to make FHE more efficient [19,21,23,32]. Recently, a more efficient FHE scheme called leveled fully homomorphic encryption without bootstrapping is reported in [23], which has a per-gate computation time of $\Omega(\lambda)$ (where λ is the security parameter) [24]. It is more efficient than Gentry-Halevi's implementation with a per-gate computation time of $\Omega(\lambda^4)$ in [18]. We use NVIDIA GPU C2050 to accelerate Gentry-Halevi implementation, gaining about 342 times speedup for encryption, 15 times speedup for recryption and 7 times speedup for decryption as reported in [30]. In this work, we try to follow our previous step to use GPU to accelerate the leveled FHE scheme.

Contribution: In this work, we propose to accelerate the leveled FHE variant using NVIDIA GPU. In the leveled FHE scheme, the crucial operation for encryption is a large-number matrix-vector multiplication. The Chinese Remainder Theory (CRT) is applied to reduce the complexity of large-number modular multiplications. It includes three main steps. During the first step, CRT is used to decompose each large-number element into many small words, which is called the decompose process. The decompose process can be precomputed in the CPU. The second process is vector operation that performs modular multiplications and additions of all these small words. Finally, the final results can be reconstructed in the reconstruction process. In our observation, the vector operation process takes most of the computation time in CPU. So we implement this part in GPU while other computations remain in the CPU. CUDA program [33] is developed to accelerate the computations by running it

in many threads in parallel on a large number of cores available on GPU. In the GPU implementation, we manage to overlap the calculation process and data transfer process to improve the computation efficiency. Experimental results show the proposed CRT-based method with GPU implementation gains about 273.6 times speedup when compared with the NTL library function and 35.2 times speedup when compared with the same CRT-based method on CPU.

1.2.4 Explore the Feasibility of FFT Multiplication for RSA Cryptosystem

Motivation: The RSA [6] cryptosystem has wide applications. With the computing technology continues to develop, it becomes necessary to upgrade the key size to 2048, 4096 or even 8192 bits to provide a higher level security although the key size with 1,024 bits is still used now. Traditionally, the interleaved Montgomery's multiplication algorithm [34] is used for the hardware design of modular multiplication in RSA cryptosystem. Since we applied the FFT multiplication algorithm for the large-number multiplication in FHE scheme and gained a very good performance in our previous work, in this work we try to use the FFT multiplication for the hardware design of RSA cryptosystem to explore its feasibility for the large-size RSA hardware design.

Contribution: In this work, we employ a novel approach for modular multiplication by combining the Strassen algorithm and Montgomery reduction [34]. Several strategies are adopted to optimize the multiplication algorithm and support efficient hardware design. The proposed design can support 8K and 12K RSA and outperform the other designs. The design can complete one 8K- and 12K-bit RSA operation in

0.104 s and 0.156 s operating at 320 MHz, which is the fastest design to the best of our knowledge. This work appears in the proceeding of 2013 IEEE HPEC [35].

1.3 Outline

This dissertation is organized as follows.

Chapter 2 introduces the fully homomorphic encryption, expecially the Gentry-Halevi's FHE scheme and BGV leveled FHE scheme. After that, the RSA cryptosystem is present.

Chapter 3 introduces the number theory arithmetic used for accelerating FHE scheme and RSA cryptosystem, including modular arithmetic, Montgomery arithmetic and FFT multiplication.

Chapter 4 presents the implementation of using GPU to accelerate Gentry-Halevi's FHE scheme. We present the optimizations method for GPU acceleration and give the performance evaluation and experimental results.

Chapter 5 describes the hardware design of a large-number multiplier for fully homomorphic encryption. We show the architecture of the VLSI design of the finite-field FFT engine and the multiplier. Finally, we give results based on VLSI synthesis and simulation results.

Chapter 6 presents the acceleration of the leveled FHE scheme using GPU. The CRT-based method and CPU implementation is described. Specifically, the GPU is used to accelerate the vector operation process, the most computation-intensive part. Finally, we give the evaluation and experimental results.

Chapter 7 presents the hardware design for RSA cryptosystem using FFT multiplication. We introduce the Montgomery modular multiplication using FFT multi-

plication algorithm, followed by the VLSI architecture of the modular multiplication. Then we present the modular exponentiation algorithm for RSA exponentiation. Finally, we give the experimental results of hardware implementation.

Chapter 8 draws the conclusions and discusses future work.

Chapter 2

Cryptographic Algorithms

In this chapter, we first introduce the two fully homomorphic encryption schemes in Section 2.1, followed by the RSA encryption in Section 2.2.

2.1 Fully Homomorphic Encryption

In the past decade, one of the most significant advances in cryptography has been the introduction of the first fully homomorphic encryption scheme by Gentry [15]. This advance not only resolved an open problem posed by Rivest [4], but also opened the door to many new applications. Indeed, using a FHE one may perform an arbitrary number of computations directly on the encrypted data without revealing of the secret key. Thus an untrusted party, such as a remotely hosted server, may perform computations on behalf of the owner on the data without compromising privacy. This property of FHE is precisely what makes it invaluable for the cloud computing platforms today. For instance, it was recognized early in [15] that the privacy of sensitive data on cloud computing platforms are ideally suited to be protected using

FHE. Considering this model of savings in scale and the recent trend, it is safe to state that cloud computing will have a significant transforming effect on business and personal computing in the coming years. This presents a perfect application target for FHE schemes.

Informally a homomorphic encryption scheme refers to an encryption function that allows one to induce a binary operation on the plaintexts while only manipulating the ciphertexts without the knowledge of the encryption key: $E(x_1) \star E(x_2) = E(x_1 \otimes x_2)$. If the scheme supports the homomorphic computation of any efficiently computable function, it is called a fully homomorphic encryption scheme. With FHE, an honest but curious party can perform any computation directly with encrypted result without gaining access to the plaintext.

The first implementation of a FHE variant was proposed by Gentry and Halevi [18], which presented an impressive array of optimizations in order to reduce the size of the public-key and to reduce the latencies of the primitives. Still, encryption of one bit takes more than a second on a high-end Intel Xeon based server, while recrypt primitive takes nearly half a minute for the lowest security setting. Furthermore, after every few bit-AND operations a recrypt operation needs to be applied to reduce the noise in the ciphertext to a manageable level. In our early work, we are trying to use GPU and design specific hardware module to accelerate the Gentry-Halevi's implementation. After Gentry-Halevi's implementation, many new FHE constructions and optimizations are developed. Especially the BGV leveled FHE [23] scheme outstands itself among these schemes as mentioned before. So we begin to use GPU to accelerate the BGV leveled FHE scheme.

2.1.1 The Gentry-Halevi FHE Scheme

We present a high-level overview of the primitives and the details can be referred to the original reference [18].

Encrypt: To encrypt a bit $b \in \{0, 1\}$ with a public key (d, r). Encrypt first generates a random "noise vector" $u = \langle u_0, u_1, \ldots, u_{n-1} \rangle$, with each entry chosen as 0 with the probability 0.5 and as ± 1 with probability 0.25 each. Then the message bit b is encrypted by computing

$$c = [u\,(r)]_d = \left[b + 2 \sum_{i=1}^{n-1} u_i r^i \right]_d \tag{2.1}$$

where d and r is part of the public key.

Eval: When encrypted, arithmetic operations can be performed directly on the ciphertext with corresponding modular operations. Suppose $c_1 = \mathsf{Encrypt}(m_1)$ and $c_2 = \mathsf{Encrypt}(m_2)$, we have:

$$\mathsf{Encrypt}(m_1 + m_2) = (c_1 + c_2) \bmod d$$
$$\mathsf{Encrypt}(m_1 \cdot m_2) = (c_1 \cdot c_2) \bmod d\ .$$

Decrypt: The encrypted bit c can be recovered by computing

$$m = [c \cdot w]_d \bmod 2 \tag{2.2}$$

where w is the private key and d is part of the public key.

Recrypt: Briefly, the Recrypt process is simply the homomorphic decryption of the

ciphertext. However, due to the fact that we can only encrypt a single bit and only a limited number of arithmetic operations can be evaluated, we need an extremely shallow decryption method. In [18], the authors discussed a practical way to re-organize the decryption process to make this possible.

Informally, the private key is divided into s pieces that satisfy $\sum^s w_i = w$. Each w_i is further expressed as $w_i = x_i R^{l_i} \bmod d$ where R is some constant, x_i is random and $l_i \in \{1, 2, \ldots, S\}$ is also random. The recrypt process can then be expressed as:

$$\begin{aligned} m &= [c \cdot w]_d \bmod 2 \\ &= \left[\sum^S c x_i R^{l_i}\right]_d \bmod 2 \\ &= \left[\sum^S c x_i R^{l_i}\right]_2 - \left[\left\lfloor (\sum^S c x_i R^{l_i})/d \right\rfloor \cdot d\right]_2 \\ &= \left[\sum^S c x_i R^{l_i}\right]_2 - \left[\left\lfloor \sum^S (c x_i R^{l_i}/d) \right\rfloor\right]_2 . \end{aligned}$$

The Recrypt process can then be divided into two parts. First compute the sum of $cx_iR^{l_i}$ for each "block" i. To further optimize this process, encode l_i to a $0-1$ vector $\{\eta_1^{(i)}, \eta_2^{(i)}, \ldots, \eta_n^{(i)}\}$ where only two elements are "1" and all other elements are "0"s. Suppose the two positions are labeled as a and b. We write $l(a,b)$ to refer to the corresponding value of l. Alternatively we can obtain $cx_iR^{l_i}$ from

$$cx_iR^{l_i} = \sum_a \eta_a^{(i)} \sum_b \eta_b^{(i)} cx_iR^{l(a,b)} .$$

Obviously, only when $\eta_a^{(i)}$ and $\eta_b^{(i)}$ are both "1", the corresponding $cx_iR^{l(a,b)}$ is selected out. In addition, if we encode the l in the way that each iteration will increase it by one, the next factor $cx_iR^{l(a,b)}$ can be easily computed by multiplying R to the result

of the previous computation.

After applying these modifications, all operations involved in this formulation of decryption become bit operations realizable by sufficiently shallow circuits. Thus we can evaluate this process homomorphically. The parameters η_i will be stored in encrypted form and incorporated into the public key.

In this scheme, for the small setting, the public key size is about 785,000 bit as reported in [18]. All the operations in the scheme is based on modular arithmetic. Compared with modular multiplication, modular addition and subtraction only has a very small computation complexity. Thus accelerating the modular multiplication becomes our first target for the Gentry-Halevi's implementation.

2.1.2 Basic Leveled FHE Encryption Scheme

The basic leveled FHE encryption scheme works as follows [23].

1. E.Setup (1^λ, 1^μ, b): λ is the security parameter, representing 2^λ security against known attacks. Use the bit $b \in \{0,1\}$ to select the parameters between a LWE-based scheme and RLWE-based scheme. Choose a μ-bit modulus M and choose the parameters $d = d(\lambda, \mu, b)$, $n = n(\lambda, \mu, b)$ and $\chi = n(\lambda, \mu, b)$ appropriately.

2. E.SecretKeyGen ($params$): Sample $\mathbf{s'} \leftarrow \chi^n$. Set $sk = \mathbf{s} \leftarrow (1, \mathbf{s'}[1], ..., \mathbf{s'}[n]) \in R_M^{n+1}$, which $R = R(\lambda)$ be a ring.

3. E.PublicKeyGen ($params$, sk): Generate $(n+1)$-column matrix $\mathbf{A}$' $\leftarrow R_M^{N \times n}$ uniformly and a vector $\mathbf{e} \leftarrow \chi^{\mathbf{N}}$ and set $\mathbf{b} \leftarrow \mathbf{A's'} + 2\mathbf{e}$. Set the public key $pk = \mathbf{A}$.

4. E.Enc($params$,pk,m): Assume the plaintext space is $R_2 = R/2R$. To encrypt

a message $m \in R_2$, set $\mathbf{m} \leftarrow (m, 0, ..., 0) \in R_M^{n+1}$, sample $\mathbf{r} \leftarrow R_2^N$ and output the ciphertext $\mathbf{c} \leftarrow \mathbf{m} + \mathbf{A}^T\mathbf{r} \in R_M^{n+1}$.

5. E.Dec$(params, pk, m)$: Output $m \leftarrow [[\langle \mathbf{c}, \mathbf{s} \rangle]_M]_2$.

In this scheme, the modulus, which is part of public key, is an odd number from 512 to 2,048 bits. As shown above, the crucial part in this scheme is a large-number matrix-vector modular multiplication with the dimension from 9,326 to 61,376. Similar to the modular exponentiation used for RSA cryptosystem, the matrix-vector modular multiplication is also needed to recursively perform modular multiplications. As a result, the modular multiplication plays a crucial part in this scheme.

2.2 The RSA Cryptosystem

The RSA cryptosystem was proposed by Rivest, Shamir and Adleman in 1978. The encryption and decryption of RSA cryptosystem are both modular exponentiation. The modular multiplications are recursively performed to finish one modular exponentiation. Usually the modulus in the RSA cryptosystems are 1,024 bits or even higher so many modular multiplications are performed for one exponentiation. As a result a fast modular multiplication is a crucial part for the real-time RSA encryption and decryption.

Assume a private key S and public key (E, M) are generated from the key generation procedure. In the RSA cryptosystems, the public key is used for encryption and private key are used for decryption. For instance, Alice has the private key and public key. Bob can use Alice's public key (E, M) to encrypt a plaintext message P to send to Alice using encryption procedure. To encrypt the message P, the plaintext

is needed to be partitioned into a sequence of blocks with each block to be an integer between 0 and $M - 1$.

$$C = P^E \bmod M$$

After Alice receives the encrypted message C, she can use the private key S to recover the message Bob original send.

$$P = C^S \bmod M$$

Chapter 3

Arithmetic

In this chapter, the modular arithmetic is present in Section 3.1. In Section 3.2, we introduce different large-number multiplications. In Section 3.3, we present the finite field FFT multiplication used in this dissertation. Then give the comparison results between different algorithms in Section 3.4.

3.1 Modular Multiplication

From last chapter, we can find the modular multiplication is a crucial part for both FHE and RSA encryptio schemes. The modular multiplication consists of the inherent multiplication and division operation, making it to be a very complicated arithmetic operation. There are usually two main method for modular multiplication. One is to perform the multiplication followed by modular reduction. Another approach is to interleave the multiplication and modular reduction when using Montgomery multiplication. For modular reduction, Montgomery reduction [34] and the Barrett reduction algorithms [36] are among the most popular modular reduction algorithms. Now the

Algorithm 3.1 Barrett Reduce Algorithm

Procedure: $r = t \bmod M$
Precomputation: $q = \lceil \log_2(M) \rceil$,$\mu = \left\lfloor \frac{2^{2q}}{M} \right\rfloor$
Process:
$r = t - \lfloor t\mu / 2^q \rfloor M$;
while $(r \geq M)$
  $r = r - M$;
end while;
return r;
end procedure

brief overview of Barrett reduction and Montgomery arithmetic are introduced.

3.1.1 Barrett Reduction

Given two positive integers t and M, the Barrett modular reduction approach computes $r = t \bmod M$. The algorithm as shown in Algorithm 3.1 requires precomputation of $\mu = \left\lfloor \frac{2^{2q}}{M} \right\rfloor$ ($q = \lceil \log_2(M) \rceil$). If multiple reductions are to be computed with the same modulo M, then this number can be reused for all reductions, which is exactly the case we have.

Note that the last step of the reduction is a loop. However, in our FHE implementation, as t is normally the result of the multiplication between two integers which are smaller than M. The loop can always finish very fast.

3.1.2 Montgomery Arithmetic

The Montgomery method was proposed in 1985 to use Residue Number System (RNS) representation of integers for modular multiplication [34]. It replaces the costly division needed for the modular reduction with shifting operations. But we need to transform the operands into the RNS domain (or Montgomery domain) and trans-

form back to get the final result.

To transform a integer x into Montgomery domain, we need to choose two coprime positive integers, the modulus N and the Radix R, $0 \leq x < N < R$. Usually the Radix R is chosen to be a powers of two to reduce the computation complexity.The Montgomery representation of x is defined by

$$\overline{x} = x \cdot R \bmod N.$$

The back transform is performed by dividing the Montgomery representation by R as follows.

$$x = \overline{x} \cdot R^{-1} \bmod N$$

The Montgomery multiplication can be performed by

$$\overline{z} = \overline{x} \cdot \overline{y} \cdot R^{-1} \bmod N.$$

There are usually two ways for Montgomery multiplication. One is to perform modular reduction after the multiplication shown in Algorithm 3.2 [34]. The another is to perform the modular reduction during multiplication, called interleaved Montgomery multiplication. The interleaved Montgomery multiplication is widely used for the small-size RSA design [28, 37] shown in Algorithm 3.3 [34]. The interleaved Montgomery multiplication with the complexity $O(N^2)$ plays a domination role in the hardware design for small-size RSA for instance 1,024-bit RSA or 2,048-bit RSA. But it may not efficient any more for the extremely large-number modular multiplication in Gentry-Halevi's scheme, compared with the method chosen an effi-

Algorithm 3.2 Montgomery multiplication

Procedure: $r = a \cdot b \cdot R^{-1} \bmod N$
Precomputation: $m = \lceil \log_2(N) \rceil$, $R = 2^m$, $N' = -N^{-1} \bmod R$,
Process:
$T = ab$;
$F = (T \bmod R)N' \bmod R$;
$r = (T + FN)/R$;
if $r \geq N$ then
$r = r - N$;
return r;
end procedure

Algorithm 3.3 Interleaved Montgomery multiplication

Procedure: $r = a \cdot b \cdot R^{-1} \bmod N$
Precomputation: $m = \lceil \log_2(N) \rceil$, $R = 2^m$, $N' = -N^{-1} \bmod R$,
Process:
$r = 0$;
for i in 0 to $k - 1$ loop
$p = r + a_i * b$;
$if(p \bmod 2 = 0)$ then $r = p/2$;
$else\ r = (p + N)/2$; end if;
end loop
if $r \geq N$ then $r = r - N$; end if;
return r;
end procedure

cient large-number multiplication algorithm. Now we are going to introduce different large-number multiplications and evaluate the performance of different algorithms.

3.2 Large Integer Multiplication Algorithms

A review of the literature shows that there is a hierarchy of multiplication algorithms. The simplest algorithm is the naive $O(N^2)$ algorithm (often called the grade school algorithm).The first improvement to the grade school algorithm was due to Karatsuba [38] in 1962. It is a recursive divide and conquer algorithm, solving an

N bit multiplication with three $\frac{N}{2}$ bit multiplications, giving rise to an asymptotic complexity of $O(N^{log_2 3})$. Toom and Cook generalized Karatsuba's approach, using polynomials to break each N bit number into three or more pieces. Once the sub-problems have been solved, the Took-Cook method uses polynomial interpolation to construct the desired result of the N bit multiplication. The asymptotic complexity of Toom-Cook algorithm depends on k (the number of pieces) and is $O(N^{log(2k-1)/log(k)})$.

The next set of algorithms in the hierarchy are based on using fast Fourier transforms (FFTs) to compute convolutions. According to Knuth [39], Strassen came up with the idea of using FFTs for multiplication in 1968, and worked with Schönhage to generalize the approach, resulting in the famous Schönhage-Strassen algorithm [40], with an asymptotic complexity of $O(N \cdot log\, N \cdot log\, log\, N)$. The FFT multiplication has the lowest computation complexity. Also it is based on FFT, which is very suitable for hardware implementation and GPU acceleration. Now we are going to give a detailed description about the FFT multiplication.

3.3 FFT Multiplication

FFT multiplication is based on convolutions. For example, to compute the product of A times B, we express the numbers A and B as sequences of digits (in some base b) and then compute the convolution of the two sequences using FFTs. Once we have the convolution of the digits, the product of A times B can be found by resolving the carries between digits. The FFT multiplication algorithm is presented as a diagram in Figure 3.1.

Briefly, the Strassen FFT algorithm can be summarized as follows:

1. Given a base b, compute the Fast Fourier Transform of the digits (with respect

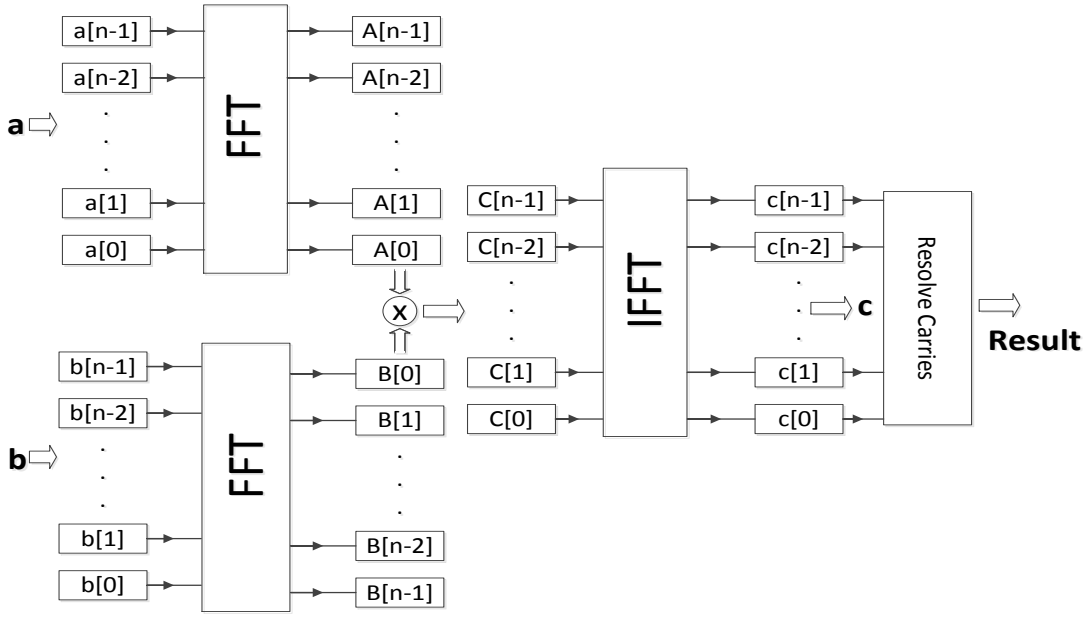

Figure 3.1: FFT-based multiplication algorithm.

to the base) of A and B, treating each digit as an FFT sample.

2. Multiply the FFT results, component by component: set $C[i] = FFT(A)[i] * FFT(B)[i]$.

3. Compute the inverse fast Fourier transform: set $C' = invFFT(C)$.

4. Resolve the carries: when $C'[i] \geq B$:set $C'[i+1] = C'[i+1] + (C'[i] \operatorname{div} b)$, and $C'[i] = C'[i] \bmod b$.

The FFT computations can done either in the domain of complex numbers or they can be done in a finite field or ring. In the complex number domain, it's trivial to construct the roots of unity required for the FFT, but the computations must be done with floating point arithmetic and the round off error analysis is quite involved. In the finite field/ring case, all the computations are done with integer arithmetic

and are exact. However, the existence and calculating the required root of unity will depend on heavily the structure of the chosen finite field/ring.

For our FFT multiplier we're going to implement the FFT in the finite field $\mathbf{Z}/p\mathbf{Z}$ where p is the prime $2^{64} - 2^{32} + 1$ [41]. This prime is from a special class of numbers called Solinas primes (see [42]). As we shall see, this choice of p has three compelling advantages for FFTs:

- We can do very large FFTs in $\mathbf{Z}/p\mathbf{Z}$. Since 2^{32} divides $p - 1$, we can do any power of two sized FFT up to 2^{32}.
- There exists a *very* fast procedure for computing x modulo p for any x.
- For small FFTs (up to size 64), the roots of unity are all powers of 2. This means that small FFTs can be done entirely with shifting and addition, rather than requiring expensive 64 bit multiplications.

3.3.1 FFTs in the Finite Field $\mathbf{Z}/p\mathbf{Z}$

To perform FFTs in a finite field we need three operators: addition, subtraction and multiplication, all modulo p, where $p = 2^{64} - 2^{32} + 1$. Addition and subtraction are straight forward (if the result is larger than p then subtract p, if the result is negative, then add p). For multiplication, if X and Y are in $\mathbf{Z}/p\mathbf{Z}$ then X*Y will be a 128-bit number, which we can represent as $X * Y = 2^{96}a + 2^{64}b + 2^{32}c + d$ (where a, b, c and d are each 32 bit values). Next, using two identities of p, namely, $2^{96} \bmod p = -1$ and $2^{64} \bmod p = 2^{32} - 1$, we can rewrite the product of $X * Y$ as:

$$\begin{aligned} X * Y &\equiv 2^{96}a + 2^{64}b + 2^{32}c + d \pmod{p} \\ &\equiv -1(a) + (2^{32} - 1)b + (2^{32})c + d \end{aligned}$$

$$\equiv \quad (2^{32})(b+c) - a - b + d$$

This means that a 128-bit number can be reduced modulo p to just a few 32-bit additions and subtractions.

Further, note that $2^{192} \bmod p = 1$, $2^{96} \bmod p = -1$, $2^{384} \bmod p = 1$, etc. This leads to a fast method to reduce any sized value modulo p. Break the value up into 96-bit chunks and compute the alternating sum of the chunks. Then reduce the result as above.

In addition to the arithmetic operator there are three other criteria in order to perform multiplication with finite field FFTs. First, to compute an FFT of size k, a primitive root of unity r_k must exist such that $r_k \hat{} k \bmod p = 1$ and $r_k \hat{} i \bmod p \neq 1$ for all i between 1 and $k-1$. Second, the value k^{-1} must exist in the field. Third, we must ensure that the convolution does not overflow, i.e., $\frac{k}{2}(b-1)^2 < p$ where k is the FFT size and b is the base used in the sampling. Finally, we must ensure that the numbers we are multiplying are less than $b^{k/2}$.

In a finite field, the process for doing an FFT is analogous to FFTs in the complex domain, thus:

$$X_i = \sum_{j=0}^{k-1} x_j (r_k)^{ij} \pmod{p} \tag{3.1}$$

And the inverse FFT is just:

$$x_i = k^{-1} \sum_{j=0}^{k-1} X_j (r_k)^{-ij} \pmod{p} \tag{3.2}$$

all of the usual methods for decomposing FFTs, such as Cooley-Tukey [43], except $(r_k)^j$ takes the place of $e^{j2\pi i/k}$.

operation	Interleaved Montgomery	Karatsuba	FFT
dmultu	302,002,176	26,138,787	2,083,530
mflo/mfhi	603,992,064	52,277,574	4,167,060
daddu/dsubu	1,207,898,112	249,505,992	49,345,563
ddrl/dsll	0	0	14,112,477
and/or	0	0	4,639,830
sltu	603,942,912	155,445,660	25,947,906
movz/movn	0	0	25,947,906
load immediate	0	0	1,886,934
TOTAL	2718.0 M	483.4 M	128.1 M

Table 3.1: Operation counts for a 786,432 bit modular multiplication

With large FFTs, the primitive roots almost always looks like random 64-bit numbers, for example, the r_{65536} that we use is *0xE9653C8DEFA860A9*. However, for FFTs of size 64 or less, the roots of unity will always be powers of 2. As we noted above, $2^{192} \bmod p = 1$ which means $(2^3)^{64} \bmod p = 1$ and therefore $r_{64} = 2^3 =$ *0x08*. Likewise, $r_{16} = 2^{12}$. This property can be used for the fast small-size FFT computation.

3.4 Modular Arithmetic Comparison

All the operations in FHE are modular operations. Usually two different approaches are used to address the modular multiplication. The first is to do multiplication first, followed by modular reduction. The other approach proposed in [34], interleaves the multiplication with modular reduction. This is an efficient grade school approach, performing the equivalent of two $O(N^2)$ multiplications. The interleaved Montgomery approach is quite commonly used for modular multiplication in the RSA algorithm, see for example [28] and [44].

To understand the arithmetic cost of different multiplication algorithms, we im-

plement three different modular multiplication algorithms in carefully tuned MIPS64 assembly and count the number of ALU operations for each. For Gentry-Halevi's scheme, the public key size is about 785,000 bit for the small setting with dimension 2,048. So the 768K-bit multiplications based on different algorithms are implemented. The first algorithm uses the interleaved version of Montgomery multiplication proposed in [34]. The second algorithm uses the non-interleaved three multiplication Montgomery reduction implemented with Karatsuba multiplication (it uses the Karatsuba method if the arguments are larger than three words, and switches to grade school multiplication to handle the base case when the arguments are small). The third algorithm adopted in this work is based on FFT multiplication and is described in detail in the next section. This algorithm also uses a traditional three multiplication Montgomery reduction. The operation counts of the three algorithms are presented in Table 3.1.

Comparing the Karatsuba and FFT multipliers, both of which compute the product and then reduce the result modulo N, we can see that FFT multiplication is faster, requiring only 1/3rd of the number of instructions as the Karatsuba multiplier. Comparing the FFT multiplier with interleaved Montgomery approach, widely used in RSA for modular multiplication, we see that the FFT multiplier uses only 1/20th of the number of instructions. The interleaved version of Montgomery multiplication is popular and efficient in RSA, but it is no longer efficient for the modular multiplication in FHE. In all, the approach we adopted for modular multiplication is the most efficient algorithm. So we choose the FFT multiplication based modular multiplication for Gentry-Halevi's scheme.

Chapter 4

Acceleration of Gentry-Halevi's Fully Homomorphic Encryption Using GPU

In 2010, Gentry and Halevi presented the first FHE implementation on an IBM x3500 server. However, this implementation remains impractical due to the high latency of encryption and recryption. The Gentry-Halevi FHE primitives utilize multi-million-bit modular multiplications and additions which are time-consuming tasks for a general purpose computer. In this work, the million-bit modular multiplication is computed in two steps, which first do a large-number multiplication followed by a modular reduction. For large number multiplication, Strassen's FFT based algorithm is employed and accelerated on a graphics processing unit through its massive parallelism. Subsequently, Barrett modular reduction algorithm is applied to implement modular reduction. As an experimental study, we implement the Gentry-Halevi primitives for the small setting with a dimension of 2048 on NVIDIA C2050 GPU. The experimental

results show the speedup factors of 7.68, 7.4 and 6.59 for encryption, decryption and recryption respectively, when compared with the existing CPU implementation.

The rest of the chapter is organized as follows. The brief introduction is present in Section 4.1. In Section 4.2, we briefly review the GPU implementation of FFT multiplication and modular multiplications. Further optimizations are discussed in Section 4.3. In Section 4.4, we present the performance evaluation and experimental results.

4.1 Introduction

Gentry-Halevi's implementation is far too inefficient for real-life employment as we mentioned above. In this work, we take another step towards this direction. We present a GPU realization of the FHE variant introduced by Gentry and Halevi [18]. Our implementation shows significant improvement in speed over the existing CPU implementation. Since GPU based cloud computing services are already available, e.g. on Amazon's EC2 cluster GPU instances, our approach is well supported on existing computing platforms. The GPU approach is also applicable from the hardware perspective. With continuous architectural improvements in recent years, GPUs have evolved into a massively parallel, multithreaded, many-core processor system with tremendous computational power. Owing to introduction of the CUDA programming paradigm, a vast of computation problems outside of the graphics domain have benefited from the superior performance of GPUs. Among the examples of the general purpose GPU computing initiative are FFT [45], data processing [46] and many other science and engineering applications [47].

An efficient modular multiplication is crucial for the FHE implementation. Many

cryptographic software implementations employ the interleaved Montgomery multiplication algorithm, c.f. [48, 49]. Montgomery multiplication replaces costly trial divisions with additional multiplications. Unfortunately, the interleaved versions of the Montgomery multiplication algorithm generates long carry chains with little instruction-level parallelism. For the same reason, it is hard to take advantage of the parallelism feature of GPUs. In [50], for example, a Montgomery multiplication implementation on NVIDIA Geforce 9800GX2 card was presented. The speedup factor of GPU decreased from 2.6 for 160-bit modular multiplication to 0.6 for 384-bit modular multiplication, which showed a negative trend with growing operand sizes. In addition, from the algorithm comparison results shown in the last chapter, we can find the FFT multiplication based modular multiplication is much more efficient than the interleaved Montgomery multiplication. As a result, the FFT multiplication is employed for our GPU acceleration.

4.2 Fast Multiplications on GPUs and Modular Reduction

The Strassen FFT Multiplication Algorithm: Large integer multiplication is by far the most time consuming operation in the FHE scheme. Therefore, it becomes the first target for optimization. As mentioned earlier, the key feature a GPU provides is parallelism. Therefore, a good parallel algorithm will be well matched with GPU hardware. In [40], Strassen described such a multiplication algorithm based on FFT, which offers a good solution for effectively parallel computation of the large-number multiplication.

Emmart and Weems' Approach: In [41], Emmart and Weems implemented the

Strassen FFT based multiplication algorithm on GPUs with computational optimizations. Specifically, they performed the FFT operation in finite field Z/pZ with a prime p to make the FFT exact. In fact, they chose the $p = 0$xFFFFFFFF00000001 from a special family of prime numbers which are called Solinas Primes [42]. Solinas Primes support high efficiency modulo computations and this p especially is ideal for 32-bit processors, which has also been incorporated into the latest GPUs. In addition, an improved version of Bailey's FFT technique [51] is employed to compute the large size FFT. Assembly language level optimization and better arrangement of shared memory for GPU cores are also introduced.

The performance of the final implementation is very promising. For the operands up to $16,320K$ bits, it shows a speedup factor of up to 19 when comparison with multiplication on the CPUs of the same technology generation. We follow their implementation in [41] and test it on the NVIDIA Tesla C2050. As we can see from Table 4.1, the actual speedup factors are slightly different from [41]. Nevertheless, it is a significant speedup over CPU. Therefore, we employ this particular instance of the Strassen FFT based multiplication algorithm in our FHE implementation.

After we have this efficient large-number multiplication algorithm, the modular reduction is the followed step that we need to focus on. Montgomery reduction [34] and the Barrett reduction algorithms [36] are among the most popular modular reduction algorithms. For the same reason as stated above, the interleaved Montgomery reduction algorithm cannot exploit the full power of the GPU. If we use large residue so that no long carry chains there, the Montgomery reduction will have the similar complexity as the Barrett reduction. However, the Barrett approach has a simpler structure and thus is easier to apply further optimizations. Therefore, we choose the Barrett method for modular reductions.

Table 4.1: Multiplication time CPU vs GPU

Size in K bits	On CPU	On GPU	Speedup
1024 x 1024	8.1 ms	0.765 ms	10.6
2048 x 2048	18.8 ms	1.483 ms	12.7
4096 x 4096	42.0 ms	3.201 ms	13.1
8192 x 8192	97.0 ms	6.383 ms	15.2
16384 x 16384	221.5 ms	12.718 ms	17.4

4.3 GPU Implementation of FHE

The FHE algorithm consists of four functions: KeyGen, Encrypt, Decrypt and Recrypt. The KeyGen is only called once during the setup phase. Since keys are generated once and then preloaded to the GPU, the speed of KeyGen is not as important. Therefore we focus our attention on the other three primitives.

For the Decrypt process, we perform the computation as in Section 2.2 of Chapter 2. Obviously, the most time consuming computation is a single operation of large number modular multiplication. Directly applying the FFT based Strassen algorithm and Barrett reduction will speed up the Decrypt operation significantly. Given that Decrypt is already sufficiently fast, we turn our attention to Encrypt and Recrypt.

4.3.1 Implementing Encrypt

For the Encrypt process, the most expensive operation is the evaluation of the degree-$(n-1)$ polynomial u at the point r. In [18], a recursive approach for evaluating the 0-1 polynomial u of degree $(n-1)$ at root r modulo d. The polynomial $u(x) = \sum_{i=0}^{n-1} u_i r^i$ is split into a "bottom half" $u^{bot}(r) = \sum_{i=0}^{n/2-1} u_i r^i$ and a "top half" $u^{top}(r) = \sum_{i=0}^{n/2-1} u_{i+d/2} r^i$. Then $y = r^{n/2} u^{top}(r) + u^{bot}(r)$ can be calculated. The degree can be repeatedly cut in half and once the degree is small enough then the "trivial implementation" can be used to compute all powers of r.

In our implementation, as the GPU does not support recursive calls, we use a more direct approach for polynomial evaluations. Specifically, we apply the sliding window technique to compute the polynomial. Suppose the window size is w and we need $t = n/w$ windows, we compute:

$$\begin{aligned}
\sum (u_i r^i) &= \sum_{j=0}^{t-1} [r^{w \cdot j} \cdot \sum_{i=0}^{w-1} (u_{i+wj} r^i)] \\
&= ((a_{t-1} r^w + a_{t-2}) r^w + a_{t-3}) r^w + \\
&\quad \ldots + a_1) r^w + a_0, \\
a_j &= \sum_{i=0}^{w-1} (u_{i+wj}),
\end{aligned}$$

where additions and multiplications are evaluated with modulo d. After organizing the computation as described above, we can introduce pre-computation to further speed up the process. As r is a known constant for the encryption, the $r^i, i = 0, 1, \ldots, w$ can be pre-computed. In addition, to reduce the overhead caused by the relatively slow communication between the CPU and the GPU, these pre-computed values can be pre-loaded into GPU memory before the Encrypt process starts. Clearly, larger window size w leads to less multiplications, which yields better performance. However, it also means higher storage requirement for more pre-computed values. Hence, it is trade-off between speed and memory use.

In our implementation, we have the dimension $n = 2048$ and $|d|$ is approximately 128KB. We can estimate the performance and storage requirement for different window sizes from that. The estimated value is listed in Table 4.2. We choose the case of window size $w = 64$ for our implementation.

Table 4.2: Comparison between different window sizes

Window Size	Number of Multiplications	Size of Pre-computed values
16	127	2 MB
32	63	4 MB
64	31	8 MB
128	15	16 MB

4.3.2 Implementing Recrypt

The Recrypt process is more complicated. As mentioned in previous section, Recrypt process can be divided into two steps: process S blocks separately and then sum them up. For the process separate blocks, the the most time-consuming computation is in the form of

$$cx_i R^{l_i} = \sum_a \eta_a^{(i)} \sum_b \eta_b^{(i)} cx_i R^{l(a,b)} \ .$$

where η_i is part of the public key. As we mentioned in previous sections, if we encode the l in a proper way such that each iteration it only increases by one, the next factor $cx_i R^{l(a,b)}$ can be easily computed by multiplying R with the result of the previous iteration. Here we refer $cx_i R^{l(a,b)}$ for each iteration as a *factor*. In each iteration, we compute $factor = factor \cdot R \bmod d$ and determine whether we should sum η_b or not. Since in this process R is a small constant, the computation may even be performed on the CPU without any noticeable loss of efficiency in the overall scheme. Therefore, the CPU is used to compute the new *factor* value while the GPU is busy computing the additions from previous iteration. This approach allows us to run the CPU and the GPU concurrently and therefore harness the the computational power in the overall system.

The constants used in Recrypt are part of the public key. They can be "pre-computed" to further speed up the process. Similar to the Encrypt, the public keys

can be pre-loaded into the GPU memory to eliminate the costly CPU-GPU communication step. Taking our implementation as an example, the public key size is about 70MB. It can perfectly fit into the 3GB GPU memory of the latest graphic cards. In fact, 3GB is enough even for the large setting in FHE [18], whose public key size is about 2.25GB.

Upon completion of processing all the "blocks", we can sum these partial results. In practice, retaining only 4 most significant bits for each number is sufficient for correctness, i.e. to make decryption work. Note that during the whole Recrypt process, all of the operations are evaluated homomorphically. All the numbers which are summed together are encrypted bit by bit. Therefore, we follow the design of binary adders and substitute bit operations with corresponding Eval operations - modular evaluation operations. The addition algorithm used here is called the grade-school addition. It takes about $O(s^2)$ multiplications to compute the sum of s numbers. Hence, we need $O(s^2)$ modular multiplications to perform the grade-school addition homomorphically. Clearly the efficiency of the Strassen-FFT and Barrett reduction based modular multiplication algorithm directly translates into an efficient homomorphic addition computation.

4.4 Experimental Results

An a case study, the Encrypt, Decrypt and Recrypt of the Gentry-Halevi FHE scheme are evaluated on a server with Intel Xeon X5650 processor running at 2.67GHz, 14 GB RAM and two NVIDIA Tesla C2050s, each of which has 448 cores, 3GB memory running at 1150MHz. However, only one GPU is used in this implementation. Shoup's NTL library [52] is used for high-level numeric operations and GNU's GMP library [53]

for the underlying integer arithmetic operations. A modified version of the code from [41] is used to perform the Strassen FFT multiplication on GPU.

For an experimental study, we employed the smallest setting with a lattice-dimension of 2,048. In this setting, the determinant d has about 790,000 bits. In practical applications, the key generation can usually be processed offline and we do not need to accelerate this part. Gentry-Halevi implementation code [18] is also executed on the the same platform for comparisons. The main results of our experiments are summarized in Table 4.3. We see that our GPU implementation is about 7.68, 7.4 and 6.59 faster for encryption, decryption and recryption, respectively, when compared to the Gentry-Halevi implementation on the CPU [18].

Table 4.3: FHE on Different Platforms

	CPU	GPU
Encrypt	1.69 sec	0.22 sec
Decrypt	18.5 msec	2.5 msec
Recrypt	27.68 sec	4.2 sec

If we look into the entire 4.2 seconds of the time it takes to compute Recrypt, we discover that it takes about 3.56 seconds to process these "blocks" and about 0.68 seconds to perform the grade-school addition. Further dissection of the block processing on GPU, about 2.66 seconds are dedicated for the multiplications and 0.24 seconds for the additions. At the same time, the CPU spends 0.9 seconds computing *factor*. Clearly, the sum of the time is more than 3.56 seconds. This indicates the fact that the CPU and the GPU are actually performing computing tasks concurrently.

4.5 Conclusions

In this chapter, we present the first GPU implementation of a fully homomorphic encryption scheme. To optimally support the higher level primitives of the Gentry-Halevi FHE, we develop efficient techniques for large integer arithmetic operations. At the lower level, we pair Emmart and Weems' implementation of Strassen's FFT multiplication with Barrett reduction to realize a high-performance modular multiplication on a GPU. Using this basic operation along with pure Barrett reduction and modular addition, we implement the FHE primitives. In addition, we tailor the encryption and recrypt functions to make optimal use of GPU features as well as to avoid obstacles, such as lack of support for recursive operations. We also develop a pre-computation strategy to further enhance the efficiency of the encryption primitive.

The performance results of the FHE primitives are obtained from the executions on a server equipped with a NVIDIA Tesla C2050 GPU. Experimental results show the speedup factors of 7.68, 7.4 and 6.59 for Encrypt, Decrypt and Recrypt, respectively, when compared with the CPU reference implementation in [18]. Although further advance are still heavily sought before FHE becomes deployable in real-world applications, this work shows that the performance of FHEs can be significantly improved by carefully choosing the target platform and by tailoring the algorithms.

Chapter 5

VLSI Design of a Large Number Multiplier for Fully Homomorphic Encryption

This chapter presents the design of a high-performance 768K-bit multiplier for fully homomorphic encryption operations. The FFT multiplication algorithm is employed for the design of the power and area efficient, high-speed multiplier. The FFT processor in the multiplier is based on a memory-based, in-place architecture to perform 64K-point finite-field FFT operations using a radix-16 computing unit and 16 dual-port SRAMs. The radix-16 calculations can be simplified to require only additions and shit operations by adopting a special prime as the base of the finite field. A two-stage carry-look-ahead scheme is employed to resolve carries and obtain the multiplication result. The proposed design has been synthesized using 90nm process technology with an estimated die area of 45.3 mm^2, which has 20.6M logic equivalent gates (two-input NAND). At 200MHz, the large number multiplier offers roughly

twice the performance of a previous implementation on an NVIDIA C2050 GPU and is 29 times faster than the Xeon X5650 CPU, while at the same time consuming a modest 0.97W.

The rest of the chapter is organized as follows: Section 5.1 gives a brief introduction to fully homomorphic encryption; Section 5.2 presents the efficient 192-bit domain operations for fast small-size FFT computation; Section 5.3 and 5.4 shows the architecture of the VLSI design of the finite-field FFT engine and the multiplier; Section 5.6 gives results based on VLSI synthesis and simulation, followed by the conclusions in Section 5.7.

5.1 Introduction and Related Work

The Gentry-Halevi scheme was the first software implementation of FHE but its computing latency is prohibitive for practical applications due to its intensive use of large-number (hundreds of thousands of bits) multiplications. In our previous work, we took Gentry and Halevi's FHE algorithm and accelerated it on a GPU platform [29] [30]. Targeted to an NVIDIA C2050 GPU with 448 cores running at 1.15 GHz, the processing time for 1-bit encryption was reduced to 45 msec and the recyption was reduced to 1.8 seconds, which is about 37.6 and 15.4 times faster than the original implementation on the CPU. Although the GPU trial provided significant acceleration, the major problem remains that the power consumption of a high-end GPU today is about 200 to 400 watts. Using GPUs to scale FHE up to data center levels is thus infeasible. The solution is to build low-power customized circuits that can provide comparable or superior performance to the fastest GPU while reducing power consumption by orders of magnitude.

Previously general-purpose GPU has also been used for acceleration of security algorithms such as elliptic curve cryptography [54]. But the GPU architecture was originally geared for graphics operations and later has been extended for general purpose computations. It is not the most power efficient architecture for a specific algorithm or applications. One approach is to attach an Application Specific Integrated Circuit (ASIC) to the CPU which is dedicated to encryption/decryption operations. At microarchitectural level, it can be implemented as an extension of instruction set. Previously customized ASIC or IP blocks has been designed to accelerate the well-known encryption schemes such as Advanced Encryption Standard (AES) and RSA [27] [28]. Today many embedded processors have AES or RSA cores included. This work is aimed to take a similar approach and to design a specific hardware or IP blocks for accelerating the core computations in FHE.

There are some works tackling the problem of hardware acceleration of fully homomorphic encryption. In [55], an FPGA implementation draft for improving the speed of FHE primitives was proposed. However, no implementation results were presented. [56] presents a first custom hardware architecture supporting encryption, decryption and recryption primitives for the lowest security setting with a dimension 2,048 for the Gentry-Halevi scheme. A number theoretical transform based fast million-bit multiplier is the heart of all the primitives as claimed in [56].

Large integer multiplication is by far the most time consuming operation in the FHE scheme. Therefore we have selected it as the first block for hardware acceleration. Because multiplication is the dominating component of FHE operations, it will be a significant step toward practical application of FHE if a high performance, low-power, area efficient, high precision, integer multiplier architecture can be developed. Therefore, our initial attempt is to tackle the design of a large-number multiplier that

can handle 768K bits, in support of the 2048 dimension FHE scheme demonstrated by Gentry and Halevi. In addition to FHE, large number arithmetic also has other important applications in science, engineering and mathematics. Specifically when we need exact results or the results that exceed the range of floating point standards, we usually turn to multi-precision arithmetic [41]. An example application is in robust geometric algorithms [57] [58] [59]. Replacing exact arithmetic with fixed-precision arithmetic introduces numerical errors that lead to non-robust geometric computations. High-precision arithmetic is a primary means of addressing the non-robustness problem in such geometric algorithms [57].

For further reading, there are a number of papers that cover hardware implementation of large number multiplication. In [60] Yazaki and Abe implement a 1024-bit Karatsuba multiplier and in [61] they investigate a hardware implementation of FFT multiplication. In [62] Kalach investigates a hardware implementation of finite field FFT multiplication. However, this work presents does not present any information about the hardware resources and performance.

For our hardware implementation, we will choose $k = 65536$ and $b = 2^{24}$. These values meet the criteria above in Chapter 3 and allow us to multiply two numbers up to $b^{k/2} = 2^{786432}$, i.e., 786432 bit in length, which is sufficient to support Gentry-Halevi's FHE scheme for the small setting with a lattice dimension of 2048.

5.2 Efficient 192-bit Wide Operations

It is often the case in our hardware FFT implementation that needs to perform a sequence of modular operations (additions, subtractions, and multiplications by powers of 2). We are going to choose a special prime as we have stated in Chapter 3

for the finite field FFT. As a result, a set of optimizations can be achieved by using the special identities of the chosen prime, including high efficient modular multiplications and small-size FFT computations. We have present the high efficient modular multiplication procedure. Now we are going to introduce the optimizations of the small-size FFT computation for the hardware design.

If we were to implement this as 64-bit wide operations, we would need to reduce the result modulo p between each stage of the pipe. Although the process to reduce a value modulo p is quite fast it still requires a lot of hardware. It turns out that if we extend each 64-bit value to 192-bits (by padding with zeros on the left) and run the pipeline with 192-bit wide values, then we can avoid the modulo p operations after each pipeline stage by taking advantage of the fact that $2^{192} \bmod p$ is 1. We do this as follows:

Addition: Suppose we wish to compute $x + y$. There are two cases, if we get a carry out from the 192^{nd} bit, then we have $\text{trunc}(x + y) + 2^{192}$ which is the same as $\text{trunc}(x + y) + 1$ modulo p (where *trunc(z)* returns the least significant 192 bits of z). If it didn't carry out, then the result is just $x + y$. We can implement this efficiently in hardware using circular shifting operations.

Multiplication by a power of 2: First let's consider multiplication by 2. Suppose we have a 192 bit value x and we wish to compute $2x$. There two cases. If the most significant bit of x is zero, then we simply shift all one bit to the left. If the top bit is set, then we need to compute $\text{trunc}(2x) + 2^{192}$ which is the same as $\text{trunc}(2x) + 1$ modulo p. In both case, it's just a left circular shift by 1 bit. Thus to compute $2^j * x$, we simply do a left circular shift by j bits.

Subtraction: Since $2^{96} \bmod p = -1$, we can simply rewrite $x - y$ as $x + 2^{96}y$. The

2^{96} is a constant shift.

For the final reduction from 192 bits back down to 64 bits, as above, we can represent a 192 bit number z as $z = 2^{160}a + 2^{128}b + 2^{96}c + 2^{64}d + 2^{32}e + f$ where a, b, c, d, e and f are each 32 bits:

$$\begin{aligned} z &\equiv 2^{160}a + 2^{128}b + 2^{96}c + 2^{64}d + 2^{32}e + f \\ &\equiv -(2^{32}-1)a - 2^{32}b - c + (2^{32}-1)d + 2^{32}e + f \\ &\equiv (2^{32}e + f) + (2^{32}d + a) - (2^{32}b + c) - (2^{32}a + d) \end{aligned} \tag{5.1}$$

5.3 VLSI Design of the Large Number Multiplier

For high throughput applications, a pipelined FFT architecture is often used [63]. However, the pipelined design requires a memory buffer at every stage [63], which becomes problematic in the context of large-integer operations. For a 64K FFT and 64 bits per data sample, we would need 4 Mbits of memory after each stage. Generally a large FFT involves numerous stages, which makes the total area for memory too large to be considered for hardware implementation.

In contrast to the pipelined FFT design, a memory-based FFT architecture adopts an in-place strategy, which allows us to store the intermediate results into the same memory as the input data. Doing so effectively minimizes the memory requirement for the FFT computation [64]. To improve throughput, multiple memory banks can be used for parallel access. In our 64K FFT architecture, a total of 16 dual-port memory banks are used and each memory bank is 256 Kbits in size. Fundamentally, the 64K FFT is implemented using four stages of 16-point FFTs. The basic concept of a stage is to perform 4096 16-point FFTs, followed by application of twiddle factors

and then transposition. If we repeat that process four times ($16^4 = 64K$), then the result is a 64K FFT. Using an in-place memory-based design, these four stages are computed sequentially using the same hardware unit and memory.

5.3.1 Radix-16 FFT Unit

One of the key elements of our design is a high throughput 16-point FFT engine. As discussed in Section 3.3, for small ($k \leq 64$) FFTs, the root of unity will always be a power of 2.

In a finite field based on the Solinas prime p, a 16-point FFT can be performed using just shift and modulo addition operations. A 16-point FFT can be expressed as as (5.2), noting $4096^{16} \bmod p = 2^{192} \bmod p = 1$. As discussed above, for 192-bit operations, any carry-out bit can be simply routed back as a carry-in bit, which is particularly suitable for hardware design.

$$X(k) = \sum_{n=0}^{15} x(n) 2^{12 \cdot nk \% 192} \bmod p \tag{5.2}$$

$$x(n) = \frac{1}{16} \sum_{k=0}^{15} X(k) 2^{(192-12nk) \% 192} \bmod p \tag{5.3}$$

For 192-bit addition, a traditional ripple-carry adder would generate a long carry chain and slow the clock speed considerably. Thus we employ carry-save adders as the basis for our high-speed design. Given three n-bit numbers a, b and c, the carry-save approach produces a partial sum ps and a shift-carry sc, where $ps_i = a_i \oplus b_i \oplus c_i$ and $sc_i = \text{BarrelLeftShifter}((a_i \wedge b_i) \vee (a_i \wedge c_i) \vee (b_i \wedge c_i), 1)$. We can cascade two 3-input

carry-save adders to form a 4-input adder. A diagram of the sum-16 unit is shown in Fig. 5.1. The summation unit is a pipeline architecture that takes 16 inputs every clock cycle. A normalization unit at the end performs a modulus p operation shown in (7) and converts the 192-bit result back to 64-bits.

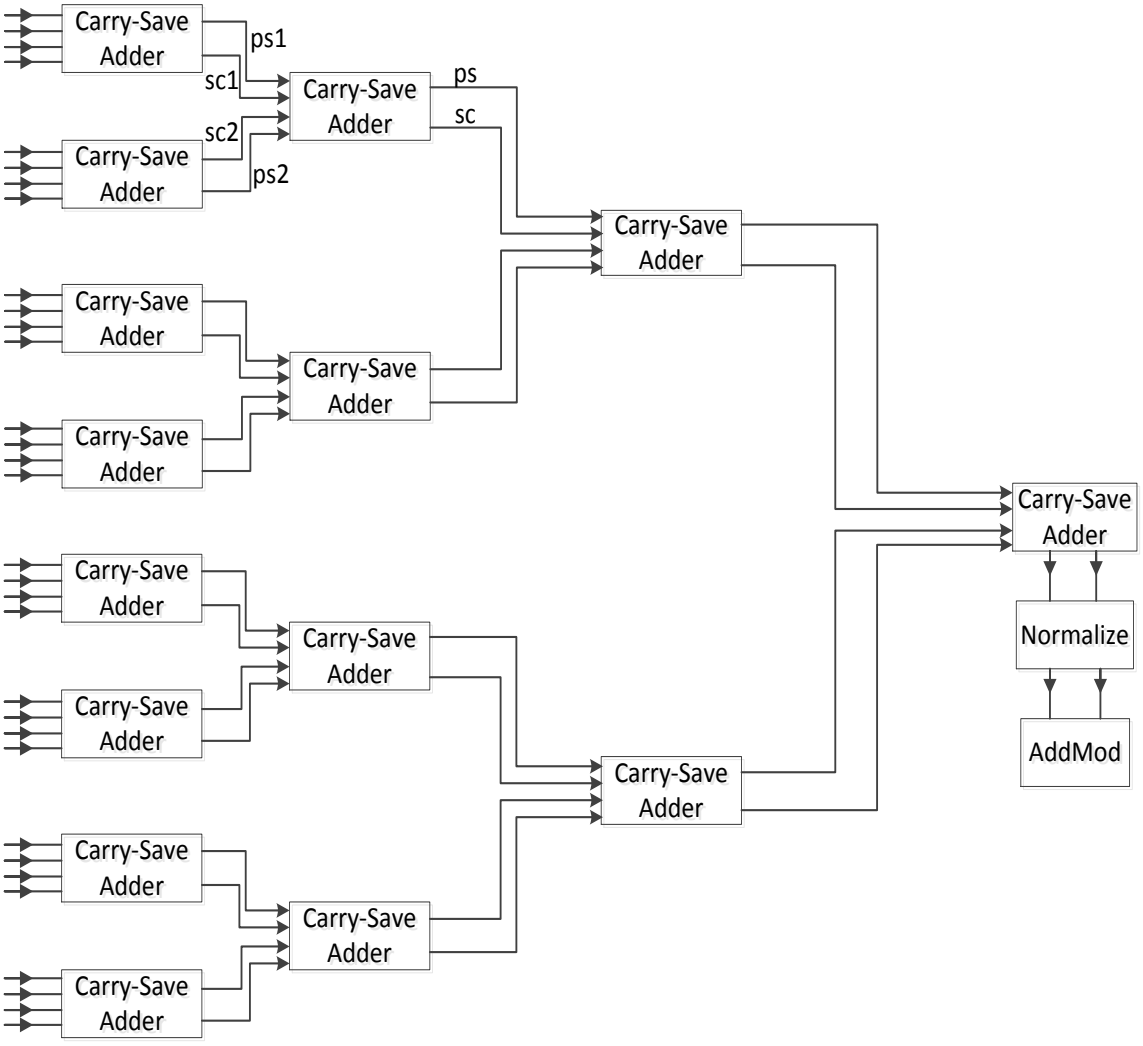

Figure 5.1: Diagram of sum-16 unit.

The architecture for a radix-16 finite field FFT unit is shown in Fig. 5.2. It consists of 16 shifters and 16 summation units. At each clock cycle, the radix-16

unit takes 16 data inputs and outputs the 16-point FFT results after a few cycles of pipeline delay.

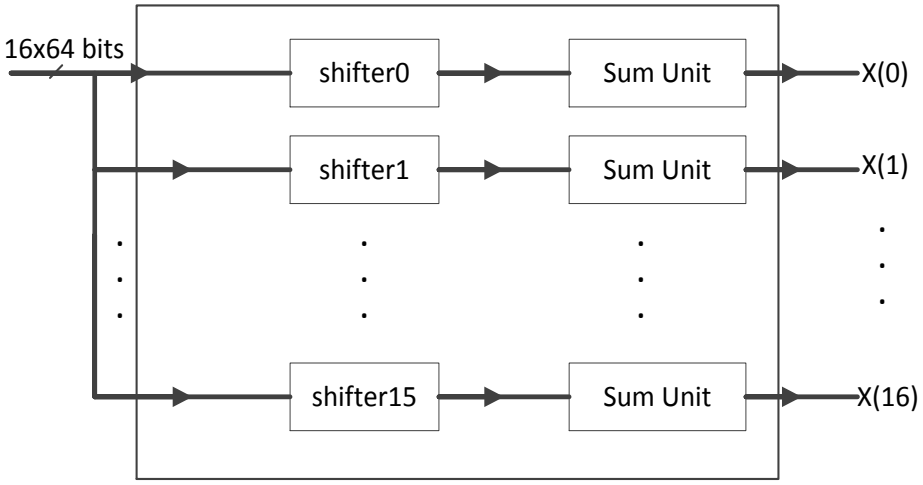

Figure 5.2: Architecture of the radix-16 FFT unit.

5.3.2 64K-Point FFT Processor

The 64K-point FFT can be decomposed into 4 stages of 16-point FFTs. At each stage, a total of 64K samples are processed through the radix-16 FFT unit. At 16 samples per cycle, that gives a total of 4096 cycles per stage. This architecture reads 16 input values from memory and writes 16 output values to the memory every clock cycle. Therefore, the memory needs to be partitioned into 16 banks. An in-place memory addressing scheme is applied to ensure there is no memory access conflict. In reference to the derivation in [64] [65], a conflict-free, in-place scheme for radix-16 FFT can be described as follows.

$$DataCount = [d_{15}, d_{14}, \ldots, d_0] \tag{5.4}$$

$$BankNum = ([d_{15}, d_{14}, d_{13,}d_{12}] + [d_{11}, d_{10}, d_9, d_8] \\ [d_7, d_6, d_5, d_4] + [d_3, d_2, d_1, d_0]) \bmod 16 \quad (5.5)$$

$$Address = [d_{15}, d_{14}, \ldots, d_4] \quad (5.6)$$

DataCount denotes the original address of the input data sample. *BankNum* is the corresponding bank assignment after partitioning. *Address* is the new address in the assigned bank. For 64K samples, the memory is partitioned into 16 banks and each bank has 4,096 samples. The data storage pattern in the memory banks is shown in Fig. 5.3.

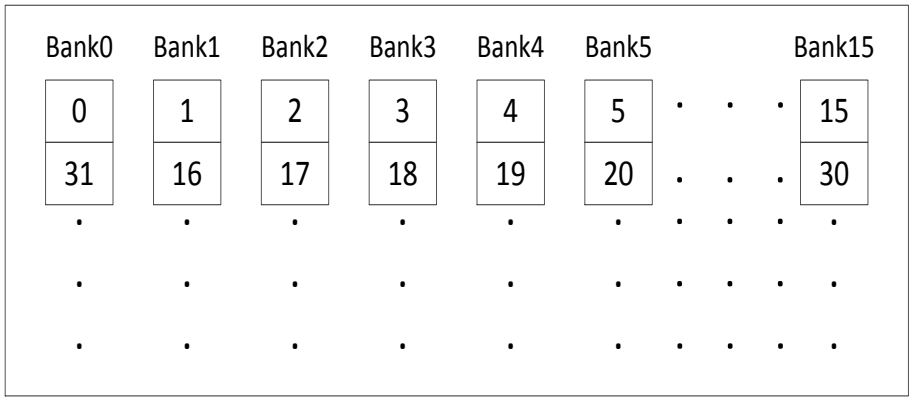

Figure 5.3: The data storage pattern in the memory banks.

The overall architecture of the FFT processor is shown in Fig. 5.4. Before entering the processor, the data has been reshuffled according to (11) and (12). The Address Generation Unit generates the corresponding bank number and address for each data sample. After all 64K samples have been received and stored in the memory banks,

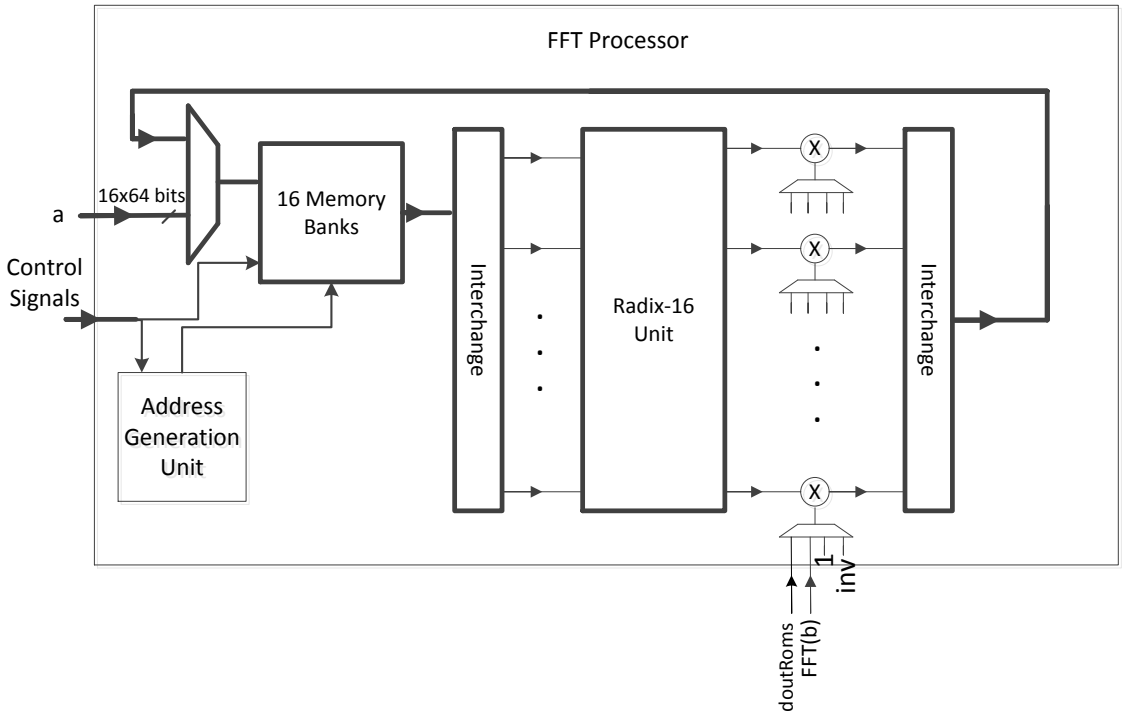

Figure 5.4: Architecture of the 64K-point FFT processor.

the FFT processor begins the computation. At each clock cycle, it reads 16 samples from the memory banks according to the *BankNo* and *Address* generated by the Address Generation Unit. These 16 values are then permuted into a proper order by the Interchange Unit and fed to the Radix-16 Unit. Subsequently, the radix-16 FFT results are modular multiplied with twiddle factors supplied from ROMs. The final results of each stage are permuted to the desired order before being stored back into the memory banks.

The modular multiplier is designed as shown in Fig. 5.5. The 64-bit multiplier has 4 pipeline stages. The 128-bit multiplication result is then split into four 32-bit components a, b, c and d. After going through the addition, shifting and subtraction as in Fig. 5.5, a 64-bit modular multiplication result is obtained.

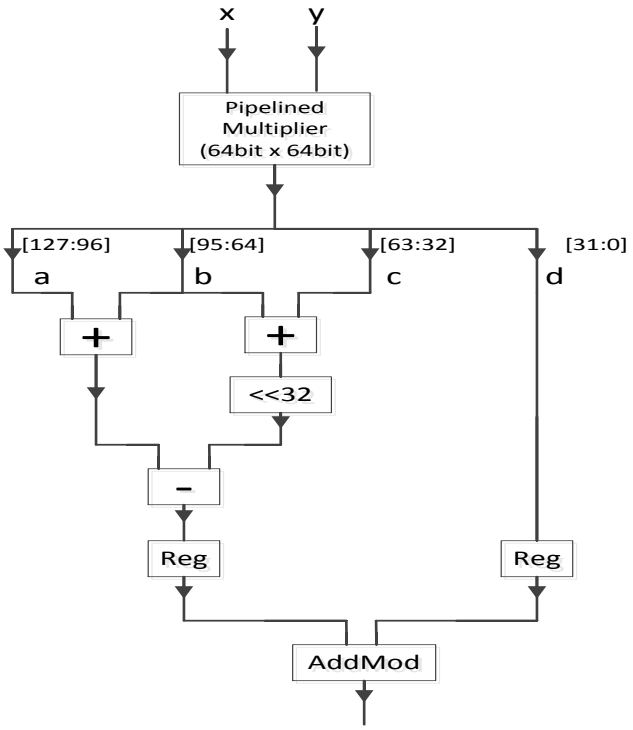

Figure 5.5: Architecture of modular multiplication unit.

5.4 Large-Number Multiplier

The high-level architecture of the large-number multiplier is shown in Fig. 5.6. It consists of two FFT processors for computing the FFTs of the two inputs a and b. Then a component-wise product is performed on the two FFT results. Subsequently, we reuse one of the FFT processors to perform the IFFT operation. The operations in each step are described as follows:

1. Data Input: The input data samples from a and b are reshuffled and stored in the corresponding addresses in the memory banks.

2. FFT: Two 64K-point FFT processors are used in the architecture. To reduce the hardware needed, both FFT processors share the twiddle factor ROMs. They also share the control signals generated by the Controller.

3. Component-Wise Product: For the point-wise product, we reuse the modular multipliers in the FFT processor. Specifically at the 4th stage of *FFT(a)*, instead of multiplying by constant 1, the result of *FFT(b)* is fed to the modular multipliers. Effectively this computes the component-by-component point-wise product of *FFT(a)* and *FFT(b)*. We thus avoid adding another set of multipliers into the design and thereby save chip area.

4. IFFT: One of the FFT processors is reused for the IFFT computation. This reuse effectively saves about 1/3 of the chip area.

5. Resolve Carries: A customized Resolve Carries unit produces the final result of large-number multiplication.

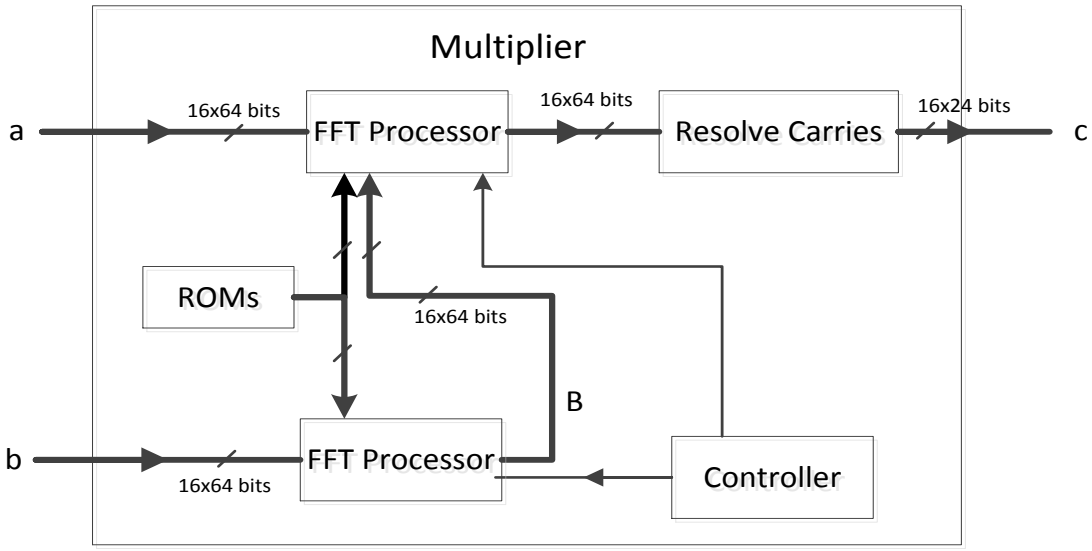

Figure 5.6: Architecture of the large-number multiplier.

5.5 Resolve Carries

To further explain the process of resolving carries, we take the 768 Kbit Strassen's multiplier as an example. But note that the design approach is general. We first decompose each 768 Kbit multiplicand into 32K groups of 24-bit numbers. Each 24-bit number is then extended to a 64-bit data sample. Owing to the convolution property of multiplication, the multiplication results are expected to be 64K groups of 24-bit numbers, or up to 1,536 Kbits, which leads to the 64K FFT in the design. Following Strassen's algorithm, the IFFT output is 64K samples of 64-bit data. The Resolve Carries unit must then obtain the actual 1,536 Kbits results from the IFFT output data.

Since each group of data is supposed to be 24-bits, each 64-bit value in the IFFT output is actually overlapped by 40-bits with the next value. For our design, we extend

the 64-bit numbers into a 72-bit format having three blocks of 24-bit numbers. The alignment among the words is illustrated as in Fig. 5.7.

Recall that the IFFT module outputs 16 data samples per clock cycle. A total of 64K data values are output in 4,096 consecutive cycles. Therefore, we must resolve the carries quickly to match the pipeline throughput. A traditional ripple-carry adder is again too slow to add 16 numbers in a row. Thus, a hierarchical carry-look-ahead scheme is employed as in Fig. 5.7. The algorithm has two steps. It first adds the words in parallel, followed by resolving the carry chain in one cycle [66]. The carry look-ahead function is shown in (13),

$$carry = ((c << 1) + carryin + critical)\ XOR\ critical \tag{5.7}$$

where $critical[i]$ and $c[i]$ are two Boolean arrays and $carryin$ is a single carry bit from the previous word. If z_i is critical ($z_i = MAX\_INT$), the i^{th} bit of $critical[i]$ is set, while the i^{th} bit of $c[i]$ is set if z_i always generates a carry ($z_i > MAX\_INT$). For a 24-bit word, MAX_INT =0xFFFFFF. For a best performance, we use a two-stage pipeline design for the Resolve Carries unit as shown in Fig. 5.7. The carry-look-ahead scheme and two stage pipeline enable the Resolve Carries unit to match the throughput of the FFT/IFFT processor output data at a high clock speed.

5.6 Experimental Results

The design of the large-number multiplier was implemented using System Verilog. The multiplier ASIC was synthesized for 90nm technology, using the Synopsys Design Compiler, the DesignWare building block libraries, and IBM 90nm CMOS 9FLP standard-cell library. Table 2 lists the synthesis results for the radix-16 unit, the 64K

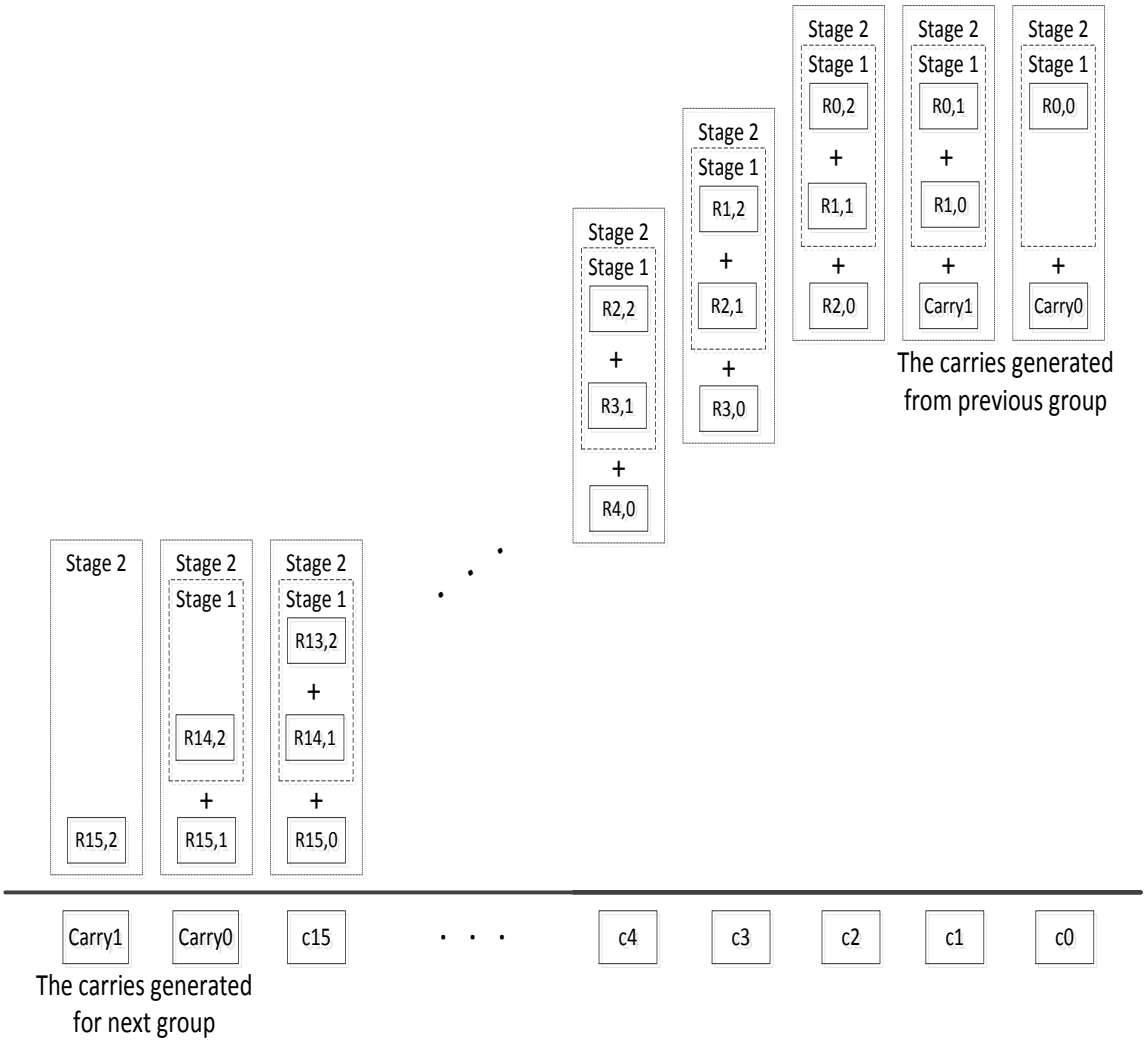

Figure 5.7: Two-stage pipeline carry resolving unit.

FFT processor, and the multiplier. The number of logic equivalent gates (two-input NAND) of the chip is 20.6M gates. A large portion of the chip area is occupied by the memories. For the large number multiplier, we have two FFT processors, each of which has 16 dual port SRAM banks of size 4,096×64 bits. The estimated area of each SRAM is about 1.07 mm^2, so the total area for the SRAMs is about 34.24 mm^2. In addition, the FFT/IFFT processors also require a set of 30 ROMs to store the twiddle factors. Each ROM is 4,096×64 bits with an estimated chip area of 0.154 mm^2. So the total area for the ROMs is about 4.63 mm^2. If combined, the total area for the RAMs and ROMs is about 38.87 mm^2, which occupies 85.8% of the chip. Thus, the architecture of the large-number multiplier is memory-constrained. In fact, the optimized radix-16 units occupy just 5% of the entire multiplier area. The proposed multiplier was also synthesized using Altera Quartus-II synthesize tool. After place and route, the design is implemented on Altera's Stratix-V 5SGXMABN1F45I2 FPGA. The resources utilized by the multiplier are listed in Table 3.

Table 5.1: Synthesis results using 90-nm CMOS technology (IBM 90nm 9FLP process)

	Radix-16 unit	FFT processor	Multiplier
Core Area	2.2 mm^2	20.7 mm^2	45.3 mm^2
Dynamic Power	313.8 mW	562.2 mW	968.7 mW
Leakage Power	25.8 uW	202.68 uW	433.1 uW
Total Power	313.83 mW	562.4 mW	969.2 mW
Clock Frequency	200 MHz	200 MHz	200MHz
Core Voltage	1.32 V	1.32 V	1.32 V

We validated the simulation results for the hardware multiplier against a software implementation using the GMP library [53]. Random numbers generated by C code were used as test vectors. The results match perfectly, thus showing that the architecture as well as the synthesized design of the large-number multiplier operate correctly.

Table 5.2: Synthesis results on Altera's Stratix-V FPGA

	Device Utilization Summary		
	Used	Available	Utilization
Combination ALUTs	243,402	718,400	34%
Dedicated logic registers	245,257	1,436,800	17%
Total block memory bits	8.912,896	54,067,200	16%
Total DSP blocks	288	352	82 %
Maximum Frequency	229.4 MHz		

Table 5.3: Performance comparison among the proposed design, CPU and GPU

	Computing Time	Speedup factor
Intel Xeon X5650 processor	6 ms	1
NVIDA Tesla C2050 GPU	0.42 ms	14.5
The proposed Multiplier	0.206 ms	29

For performance evaluation, we compare the throughput of our multiplier with the software implementations on CPU and GPU. The 768K-bit multiplication was evaluated on a high-end server with an Intel Xeon X5650 processor running at 2.67GHz with 24 GB RAM using the GMP library, which supports arbitrary precision arithmetic, and is carefully designed using fast algorithms and highly optimized assembly code, as necessary [53]. The execution time on the CPU is about 6 ms. The same Strassen's multiplication algorithm was also implemented on an NVIDA Tesla C2050 GPU, which has 448 cores running at 1.15 GHz as in [29]. It takes 0.0657 ms to transfer a 786,432-bit number from Xeon processor to the GPU or transfer a 786,432-bit number from the GPU back to the Xeon processor. When the data has been transferred to the GPU, we measure the run time of the GPU kernel, and then transfer the results back to the GPU. The GPU kernel execution time is 0.42 ms, excluding the data transfer time between CPU and GPU. For our hardware implementation, it takes 4096 cycles to load the samples into SRAMs, eight stages of FFT/IFFT with 4,119 cycles per stage, and 4,098 cycles to read the multiplication results out of the

memory. At 200 MHz, the execution time of the VLSI implementation is 0.206 ms, which is twice as fast as the GPU and 29 times faster than the CPU as listed in Table 4. More importantly, the proposed VLSI implementation uses approximately 0.97 watts, which is significantly less power than either the GPU or CPU, making it more suitable for scaling up.

For comparison, in [61], Yazaki and Abe implement a 32,768-bit FFT based multiplier in hardware, in an area of 9.05 mm^2 using a 0.18 μm process. They achieve a run time 1.02 ms for a 32,768-bit multiplication. Our multiplier handles numbers 24 times larger and at 5 times the speed. In [56], a million-bit multiplier for Gentry-Halevi FHE scheme is designed with 26.7M gates. It can finish one large-number multiplication for FHE scheme in 7.75 ms. Our design can calculate one large-number multiplication in 0.206 ms with 20.6M gates, which can be 37.6 times faster than the design in [56].

5.7 Conclusions

In this work, an efficient VLSI implementation of a large-number multiplier is presented using Strassen's FFT-based multiplication algorithm. To the best of our knowledge, this is the largest multiplier that has been implemented using VLSI design. Due to memory constraints, a memory-based, in-place FFT architecture was used for the FFT processor. A set of design optimization strategies were applied to improve the performance and reduce the are of both the Radix-16 unit and the Resolve Carries unit. The multiplier was synthesized for 90nm technology with an estimated core area 45.3 mm^2. Experimental results show that the proposed multiplier is about 2 times faster than GPU and 29 times faster than CPU, and its power consumption is

less than 1 watt.

Chapter 6

Accelerating Leveled Fully Homomorphic Encryption Using GPU

In this chapter, we try to use GPU to accelerate the large-number matrix-vector multiplication, the most crucial part of the encryption algorithm in the leveled FHE scheme. the Chinese Remainder Theorem is employed to reduce the computational complexity of the large-number element-by-element modular multiplication. The first step is called decomposition, in which each large-number element in the matrix and vector is decomposed into many small words. The next step is vector operation that performs the modular multiplications and additions of the decomposed small words. Finally the matrix-vector multiplication results can be obtained through reconstruction. We compare the CRT-based method with Number Theory Library, showing the proposed method is about 7.8 times faster when executing on CPU. In addition, it is observed that vector operation takes up to 99.6% of the total computation time

and the reconstruction only takes 0.4%. Therefore GPU acceleration is employed to speed up the vector operations. In the GPU implementation, the GPU computation and data transfer process between GPU and CPU are overlapped. Experiment results show that the GPU implementation of the CRT-based method is 35.2 times faster than the same method implemented on CPU and is 273.6 times faster than the NTL library on CPU.

The rest of the chapter is organized as follows.The brief introduction is present in 6.1. The CRT-based method and CPU implementation is described in Section 6.2. In Section 6.3, we present the method for GPU implementation. Section 6.4 gives the evaluation and experimental results.

6.1 Introduction

FHE is hard to have a practical application in real life due to its serious efficiency impediments. Several different FHE schemes has been proposed to make FHE more efficient [19, 21, 23, 32]. Recently, a more efficient FHE scheme called leveled fully homomorphic encryption without bootstrapping is reported in [23]. It has a per-gate evaluation time of $\Omega(\lambda)$ (λ is a security parameter), which is more efficient than the Gentry-Halevi implementation with a per-gate evaluation time of $\Omega(\lambda^4)$. In this chapter, we want to follow our previous step and try to use GPU to accelerate the leveled FHE scheme.

A recent work in [67] implemented the Advanced Encryption Standard homomorphically using this leveled FHE scheme, which took about 36 hours on a PC to evaluate a single AES encryption operation. It is too slow for any practical applications. In this implementation [67], for the smallest case (the depth $L = 10$) the dimension for

the public key matrix is 9326, with the modulus q an odd number ranging from 512 to 2,048 bits. As discussed above, the crucial part in encryption is a matrix-vector multiplication and decryption is actually a vector-vector multiplication. In this work, we focus on accelerating the matrix-vector multiplication which is considered the most computation intensive part in the leveled FHE encryption scheme.

6.2 Software Implementation on CPU

6.2.1 CRT Representation and Barrett Reduction

As mentioned in [67], the modulus is an odd number from 512 to 2,048 bits. For the matrix-vector multiplication, the computations are essentially large-number multiplications with each multiplicand in the size of 512 to 2,048 bits. This is similar to the modular multiplication in RSA. In this research, we choose a medium size modulus of 1,024 bits for evaluation. The CRT method has been used widely in reducing the computational complexity for RSA encryption [68]. Hereby, we propose to apply the CRT method to the element-by-element modular multiplication and addition for the matrix-vector multiplication. We can choose a special odd number for M. When CRT is applied, it can be broken into 32 coprime pairwise modulies with each 32 bits.

Initially, the 1,024-bit number is decomposed into 32 integers each with 32 bits during CRT decompose process. In the vector operation process, a modular reduction is required after each 32-bit by 32-bit multiplication. Thus an efficient modular multiplication is crucial for software implementation. Montgomery reduction [34] and the Barrett reduction algorithms [36] are the most popular modular reduction algorithms. Compared with Barrett reduction, Montgomery reduction needs extra computational steps to convert integers into Montgomery domain and later convert

Algorithm 6.1 Dot Product Using Chinese Remainder Theorem

Procedure: $\mathbf{a} \times \mathbf{b} \bmod \mathbf{M} = a_0 b_0 + a_1 b_1 + ... + a_{\mathrm{N}-1} b_{\mathrm{N}-1}$.

Decompose: Let the numbers $m_0, ..., m_{k-1}$ be positive integers which are pairwise co-prime with product $M = \prod_{i=0}^{k-1} m_i$. Thus the large numbers a_0,...,$a_{N-1}$and b_0,...,b_{N-1} can be decomposed as follows. The decompose process can be precomputed.

$a_{0,0} = a_0 \bmod m_0$, $a_{0,1} = a_0 \bmod m_1$,..., $a_{0,k-1} = a_0 \bmod m_{k-1}$
$a_{1,0} = a_1 \bmod m_0$, $a_{1,1} = a_1 \bmod m_1$,..., $a_{1,k-1} = a_1 \bmod m_{k-1}$
$\vdots \qquad\qquad \vdots \qquad\qquad \vdots$
$a_{N-1,0} = a_{N-1} \bmod m_0$, $a_{N-1,1} = a_{N-1} \bmod m_1$,..., $a_{N-1,k-1} = a_{N-1} \bmod m_{k-1}$

$b_{0,0} = b_0 \bmod m_0$, $b_{0,1} = b_0 \bmod m_1$,..., $b_{0,k-1} = b_0 \bmod m_{k-1}$
$b_{1,0} = b_1 \bmod m_0$, $b_{1,1} = b_1 \bmod m_1$,..., $b_{1,k-1} = b_1 \bmod m_{k-1}$
$\vdots \qquad\qquad \vdots \qquad\qquad \vdots$
$b_{N-1,0} = b_{N-1} \bmod m_0$, $b_{N-1,1} = b_{N-1} \bmod m_1$,..., $b_{N-1,k-1} = b_{N-1} \bmod m_{k-1}$

Vector Operations:
$t_0 = (a_{0,0}b_{0,0} + a_{1,0}b_{1,0} + ... + a_{N-1,0}b_{N-1,0}) \bmod m_0$,
$t_1 = (a_{0,1}b_{0,1} + a_{1,1}b_{1,1} + ... + a_{N-1,1}b_{N-1,1}) \bmod m_1$,
$\vdots$
$t_{\mathrm{k}-1} = (a_{0,k-1}b_{0,k-1} + a_{1,k-1}b_{1,k-1} + ... + a_{N-1,k-1}b_{N-1,k-1}) \bmod m_{k-1}$,

Reconstruction: The dot product result can be reconstructed as follows.
$\mathbf{a} \times \mathbf{b} \bmod \mathrm{M} = \sum_{i=0}^{k-1} t_i v_i M_i$,
where $M_i = M/m_i$, and $v_i = M_i^{-1} \bmod m_i$.

back from Montgomery domain. So Barrett method is employed for the modular reductions in this evaluation.

6.2.2 Software Implementation

he matrix-vector multiplication involves a set of N dot-product operations if the matrix has N columns. The decompose process using CRT can be precomputed so we exclude the execution time of the decompose process in the evaluation. We implement the matrix-vector multiplication using the CRT method using C/C++. We validate the results for our CRT implementation by comparing to the function of the large-number matrix-vector multiplication in NTL library [52]. Random numbers generated by C code are used as test vectors. From Table 6.1, it shows that the CRT method is about 7.8 faster than the function in the NTL library when both executing on a PC. Also the vector operations take about 99.6% of the total computing time in the CRT method, which is the most computation-intensive part. As a result, we propose to use GPU to accelerate the vector operations, while leaving the reconstruction process and other remaining operations in the CPU.

Table 6.1: Performance comparison among NTL and the CRT method

	Vector Operations	Reconstruction	Total
NTL library	555.4 sec	0	555.4 sec
CRT method	71.2 sec	0.343 sec	71.5 sec
Speedup	—	—	7.8

6.3 GPU Implementation

Two kernel functions are developed to implement the steps of vector operations as in Algorithm 6.1. The first kernel function is kernel_BarretMulMod( ), which computes $r = xy \bmod M$ $(x < M, y < M)$ described in Algorithm 3.1. To save memory space, the resulted matrix overwrites the input matrix since their dimensions are exactly the same. The other kernel function is kernel_addmodcal( ) used for modular additions. Both kernel functions use two-dimensional block and thread indexing, as explicitly parallel processing in the GPU.

The size of input matrix can be too large to fit into the GPU memory. For example, if the matrix has 9326×9326 elements, and each element is converted to 32 integer numbers each with 32-bits, then the memory size is 10.4 GB for this matrix only. To solve this problem, we divide the input data into several sections and keep them independent during computation. The GPU kernel function process one section of the data each time and the data of next section is transferred from host memory to GPU memory simultaneously. Thus the computation and data transfer are completely overlapped, thus the data transferring time is hidden. Based on our experiment for this particular case, this method can achieve a speedup of 1.96 in performance compared to non-overlapped GPU version shown in Table 6.2. Fig 6.1 illustrates the process of the overlapped implementation. We allocate CUDA page-locked (pinned) memory for the input data to enable asynchronous data transfers. Two CUDA streams are created: one for computing and the other for data transfer. CPU and GPU synchronization is performed at the end of each section to ensure all computation and data transfer are completed. The pointer of the used data is exchanged with that of the newly imported data, making the two memory blocks

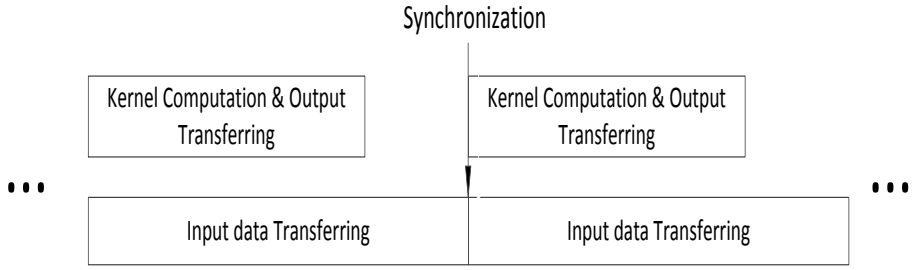

Figure 6.1: Overlapping computation and data transfer

ready for the next section.

Table 6.2: Performance comparison among overlapped GPU and non-overlapped GPU

	Vector Operations
None-overlapped GPU version	3.32 sec
Overlapped GPU version	1.69 sec
Speedup	1.96

6.4 Experimental Results

As a case study, the CRT-based matrix-vector multiplication are evaluated on a desktop computer with Intel i5 3570K processor running at 3.4 GHz, 32 GB DDR3 RAM and one NVIDIA Tesla K20, which has 2,496 cores, 5GB DDR5 memory. Shoup's NTL library [52] is used for performance comparison and result validation.

Here we employ the smallest setting in [67] with a matrix dimension of 9,326 and the size of modulus M has 1,024 bits. In the CRT-based method, each 1,024 element is first decomposed into 32-bit small words. As mentioned in Section 3, our CRT-based matrix-vector multiplication is about 7.8 faster than the NTL library function on the CPU. Since the vector operation process takes 99.6% of the total calculation time, we use GPU to accelerate this vector operation process. When implemented

on GPU, the vector operation process takes about 1.69 seconds which is 42.1 times faster than its implementation on CPU as shown in Table 6.3. We compare the NTL-based calculation time on CPU, the CRT-based method on CPU, and the CRT-based method with GPU acceleration. The results are listed in Table 6.4 and Fig. 6.2.

Table 6.3: Performance comparison of vector operation process among

	Vector Operations	Total Cal. Time
CPU	71.2 sec (CPU)	71.5 sec
CPU plus GPU	1.69 sec (GPU)	2.03 sec
Speedup	42.1	35.2

Table 6.4: Performance comparison among NTL, CRT on CPU and CRT with GPU

	Calculation Time	Speedup
NTL on CPU	555.4 sec	1
the CRT-based method on CPU	71.5 sec	7.8
the CRT-based method with GPU	2.03 sec	273.6

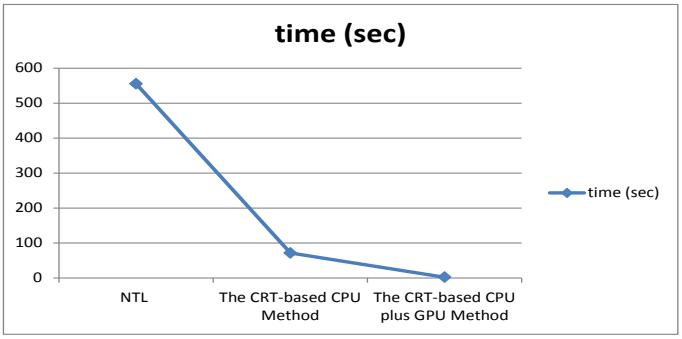

Figure 6.2: Execution time comparison

In the smallest case with the dimension of matrix 9,326 and the modulus 1,024 bits, it requires about 10.4 GB memory to store a matrix. From Table 6.5, we can see that larger memory space is needed as the matrix dimension grows. Given the

limitation of 32 GB RAM in the computer we use, only the small case is evaluated as an initial study.

Table 6.5: Memory Space in Different Settings

Matrix Dimension	Memory Space for one Matrix
9326	10.4 GB
19434	45.0 GB
29749	105.5 GB
40199	192.6 GB
50748	307.0 GB
61376	898.1 GB

6.5 Conclusion

In this work, the CRT method is used to implement the large-number matrix-vector multiplication. Compared to the NTL library function, the CRT-based method gains about 7.8 speedup when both executing on CPU. In order to further accelerate the matrix-vector multiplication, we use GPU to accelerate the vector operation process, which accounts for 99.6% of the total computation. In the GPU implementation, we manage to overlap the calculation process and data transfer process to improve the computation efficiency. Experimental results show the proposed CRT-based method with GPU implementation gains about 273.6 times speedup when compared with the NTL library function and 35.2 times speedup when compared with the same CRT-based method on CPU.

Chapter 7

Explore the Feasibility of FFT Multiplication for RSA Cryposystem

This chapter presents a novel and fast modular multiplication and exponentiation architecture for large key-size RSA cryptosystem. The Strassen multiplication and Montgomery reduction are combined for the large-number modular multiplication, which is different from the traditionally used interleaved version of Montgomery multiplication method in RSA hardware design. The proposed design can support 8,192-bit or 12,288-bit modular multiplication by selecting different bases of 16 or 24 bits. A new RSA modular exponentiation algorithm using FFT multiplication are proposed to reduce one third calculation time of the large-number multiplication in modular multiplication. The design was implemented on the Altera's Stratix-V FPGA and 90-nm application-specified integrated circuit technologies. It performs one 8K-bit modular multiplication in 6.34 μs and one modular exponentiation in 0.104 s when operating at 320 MHz.

The rest of the chapter is organized as follows: Section 7.1 gives a brief introduc-

tion of RSA design; Section 7.3 shows the VLSI architecture of the modular multiplication; Section 7.6 gives the experimental results of FPGA implementation followed by the conclusions in Section 7.7.

7.1 Introduction

The RSA [6] system is one of the most widely used public key cryptography systems. As mentioned above, the RSA operation is a modular exponentiation and its security level relies on that there are no effective procedures or algorithms that can factorize large integers within a short time period using current computer technology. Now the size of modulus is at least 1,024 bits to provide a good level of security. As the Moore law continues driving the computer technology, the key size of 1,024 bits can be broken. It becomes necessary to upgrade the key size to 2048, 4096 or even 8192 bits to provide a higher level security. It is hard to achieve a good throughput rate without the use of hardware acceleration because of computing complexity.

RSA cryptosystem recursively performs modular multiplications to finish one modular exponentiation. As a result, the performance of a RSA system relies on the throughput rate of the modular multiplication. There are two methods for modular multiplication. One is the the interleaved Montgomery's multiplication algorithm [34], and the other is to do multiplication firstly, followed by modular reduction. Traditionally, the interleaved Montgomery's multiplication algorithm with the complexity $O(N^2)$ is used to speed up the modular multiplication calculation. For the small size RSA, the interleaved Montgomery modular multiplication algorithm is a good choice that can achieve high performance at a low cost of hardware [28, 37, 69, 70]. The FFT based Strassen algorithm proposed in [40], with the

complexity of $O(N \cdot logN \cdot loglogN)$, is a high efficient large-number multiplication algorithm. The complexity of modular multiplication will be $O(N \cdot logN \cdot loglogN)$ if the FFT multiplication is used for the three large-number multiplications in modular multiplication, providing a promising option for RSA implementation with the key size growing. In this research, we employ a novel approach for modular multiplication by combining the Strassen algorithm and Montgomery reduction [34]. Several strategies are adopted to optimize the multiplication algorithm and support efficient hardware design. The proposed design can support 8K and 12K RSA and outperform the other designs even if the interleaved Montgomery algorithm is more efficient than the FFT based algorithm at these two key sizes.

7.2 Montgomery Modular Multiplication

In this chapter, we use the same FFT multiplication for the large-number multiplication in RSA design. In this implementation, we choose the base b to be 16 or 24, so every sample has 16 or 24 bits. For a total of 512 samples, we can perform 8192-bit or 12,288-bit multiplication. As we know above, the multiplication of two numbers is similar to the cyclic convolution result of two signals each with 512 samples. Typically, cyclic convolution involves "zero padding" and the result contains approximately twice many samples as that of the input signal. Thus, a high-speed 1024-point finite-field FFT processor is proposed in this design.

The most popular algorithms for modular reduction are the Montgomery reduction [34] and the Barrett reduction algorithms [36]. For the reason as stated in previous section, the interleaved Montgomery algorithm generates a long carry chain. If we use large residue without long carry chains, the Montgomery reduction has the similar

Algorithm 7.1 Montgomery Multiplication Using FFT Multiplication

Procedure Montgomery(X,Y, M): $c = XYR^{-1} (\mathrm{mod}\, M)$
Precomputation: $n = \left\lceil log_2^M \right\rceil$, $R = 2^n$,$M' = -M^{-1} (\mathrm{mod}\ R)$
1. $T \leftarrow IFFT(\ FFT(X)\ \odot\ FFT(Y)\ )$;
2. $t\ \leftarrow T \bmod R$;
3. $U \leftarrow IFFT(\ FFT(t)\ \odot\ FFT(M')\ )$;
4. $u\ \leftarrow U \bmod R$;
5. $W \leftarrow IFFT(\ FFT(u)\ \odot\ FFT(M)\ )$;
6. $C \leftarrow T + W$;
7. $c\ \leftarrow C/R$;
8. If $c \geq M$ then $c \leftarrow c - M$, end if
end procedure.

complexity as the Barrett reduction. Since it is hard to design the control logic for Barrett algorithm, we choose the Montgomery method in the hardware design.

We employ the Strassen algorithm for the calculation of the three large-number multiplications in the Montgomery multiplication as shown in Algorithm 7.1. Multiplying two numbers is equivalent to the component-wise product of the FFT results of two signals in the FFT domain. Thus, we can precompute the FFTs of M and M' to reduce the computational complexity.

7.3 VLSI Design of the Modular Multiplication

As described in Section 3.3, the finite-field FFT/IFFT is a key component for the FFT-based Strassen's multiplication algorithm. The memory-based in-place FFT architecture allows to store the intermediate results into the same memory where the input data are read from. As a result, it minimizes the memory usage while still produces high throughput [64]. In this work, we use the memory-based in-place FFT architecture and radix-32 butterfly computation. As a result, the 1024-point FFT is implemented using two stages of 32-point FFT. Using in-place memory-based design,

these two stages are computed sequentially using the same hardware unit and memory space. The radix-16 butterfly unit can be recursively used four times to complete one radix-32 FFT computation. Therefore, we employ only one radix-16 butterfly unit instead of the much larger radix-32 unit to further reduce hardware cost.

7.3.1 Radix-16 FFT Unit

With the chosen prime p, 64 is a 32^{th} root, 4096 is a 16^{th} root, 4096^2 is a 8^{th} root and so on. This means that 32-point, 16-point and 8-point FFTs can be done with shift operations rather than costly multiplications. The 16-point FFT can be simplified as (7.1), since $4096^{16} \bmod p = 2^{192} \bmod p = 1$. For 192-bit operations, any carry-out bit can be simply routed back as a carry-in bit. This special property is useful for hardware design. The multiplications in 16-point FFT can be accomplished by circular shifting operations. Instead of performing modular operations after each addition, we add all 16 numbers first and perform the modular reduction only once to obtain the final result. Since $2^{192} \bmod p = 1$, only 192 bits needs to be kept during the additions.

$$X(k) = \sum_{n=0}^{15} x(n) 2^{12 \cdot nk \% 192} \bmod p \tag{7.1}$$

$$x(n) = \frac{1}{16} \sum_{k=0}^{15} X(k) 2^{(192-12nk) \% 192} \bmod p \tag{7.2}$$

For 192-bit addition, traditional carry-ripple adder would generate a long carry chain and slow down the clock speed considerably. So we choose carry-save adders

that support high-speed design. The diagram of a processing element (PE) in radix-16 unit is shown in Fig. 7.1. At every cycle, 16 samples are read into the PE, shifted by the shifter and accumulated by the carry-save adders. At the end, a reduction unit performs modulus p operation and converts the 192-bit result back to 64-bit. Again, the special identities mentioned above are employed to simplify the calculation as shown in (3), where a, b, c, d, e and f are 32-bit components of the 192-bit result. The radix-16 unit has 16 processing elements. At each clock cycle, the radix-16 unit takes 16 data inputs and outputs the 16-point FFT results after a few cycles of pipeline delay.

$$\begin{aligned} z &= 2^{160}a + 2^{128}b + 2^{96}c + 2^{64}d + 2^{32}e + f \\ &= (2^{32}e + f) + (2^{32}d + a) - (2^{32}b + c) - (2^{32}a + d) \end{aligned} \tag{7.3}$$

7.3.2 Resolve the Carries

We take the 8192-bit Strassen's multiplier as an example to explain the process of resolving carries. Each 8192-bit multiplicand is first decomposed into 512 groups of 16-bit numbers. Then each 16-bit number is then extended to a 64-bit data sample. The multiplication results are expected to be 1024 groups of 16-bit numbers, or up to 16,384 bits. Following the Strassen's algorithm with 1024-point FFT, the IFFT output are 1024 samples of 64-bit data. The resolve carries unit is to obtain the actual 16,384-bit results from the IFFT output data.

Since each data is supposed to be 16-bit, each 64-bit data from IFFT output are actually overlapped 48-bit with the next one. For a structural design, we decompose

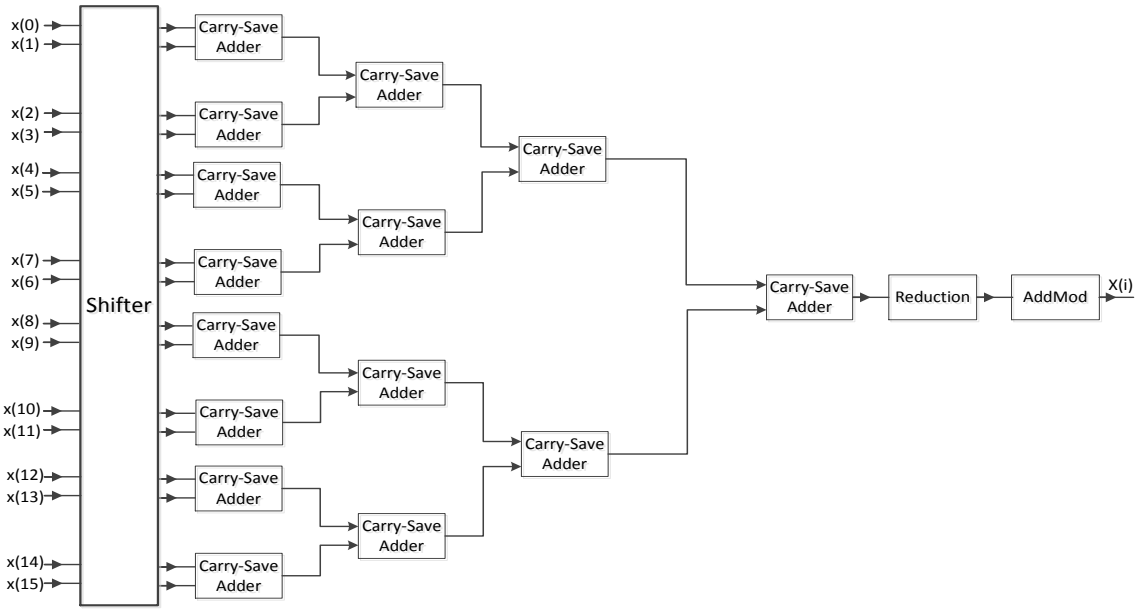

Figure 7.1: Diagram of One Processing Element.

the 64-bit number into four blocks of 16-bit words. The alignment among the words are illustrated as in Fig. 7.2.

Recall that the IFFT module outputs 32 data samples per clock cycle in operation. A total of 1024 data are output in 32 consecutive cycles. Therefore, we have to resolve the carriers in time to match the pipeline throughput. Apparently the traditional carry-ripple adder is too slow to add the numbers in a column. A hierarchical carry-look-ahead scheme for large-number addition as proposed in [66] is applied here to add the the numbers in parallel. For a high-speed design, we use a four-stage pipeline design for the resolving carries unit. Overall by using the carry-look-ahead scheme and four stage pipeline, the resolve carries unit can meet the throughput of the FFT/IFFT processor at high clock speed.

If we want to do a 12,288-bit multiplication, each multiplicand is first decomposed into 512 words each with 24 bits. Similarly, each 24-bit number is extended to a 64-

bit data sample. After processed by FFT and IFFT, the 64-bits data samples are extended into 72-bit format as three blocks of 24-bit numbers. Then we use the similar parallel and hierarchical carry-look-ahead scheme to add the numbers in each column.

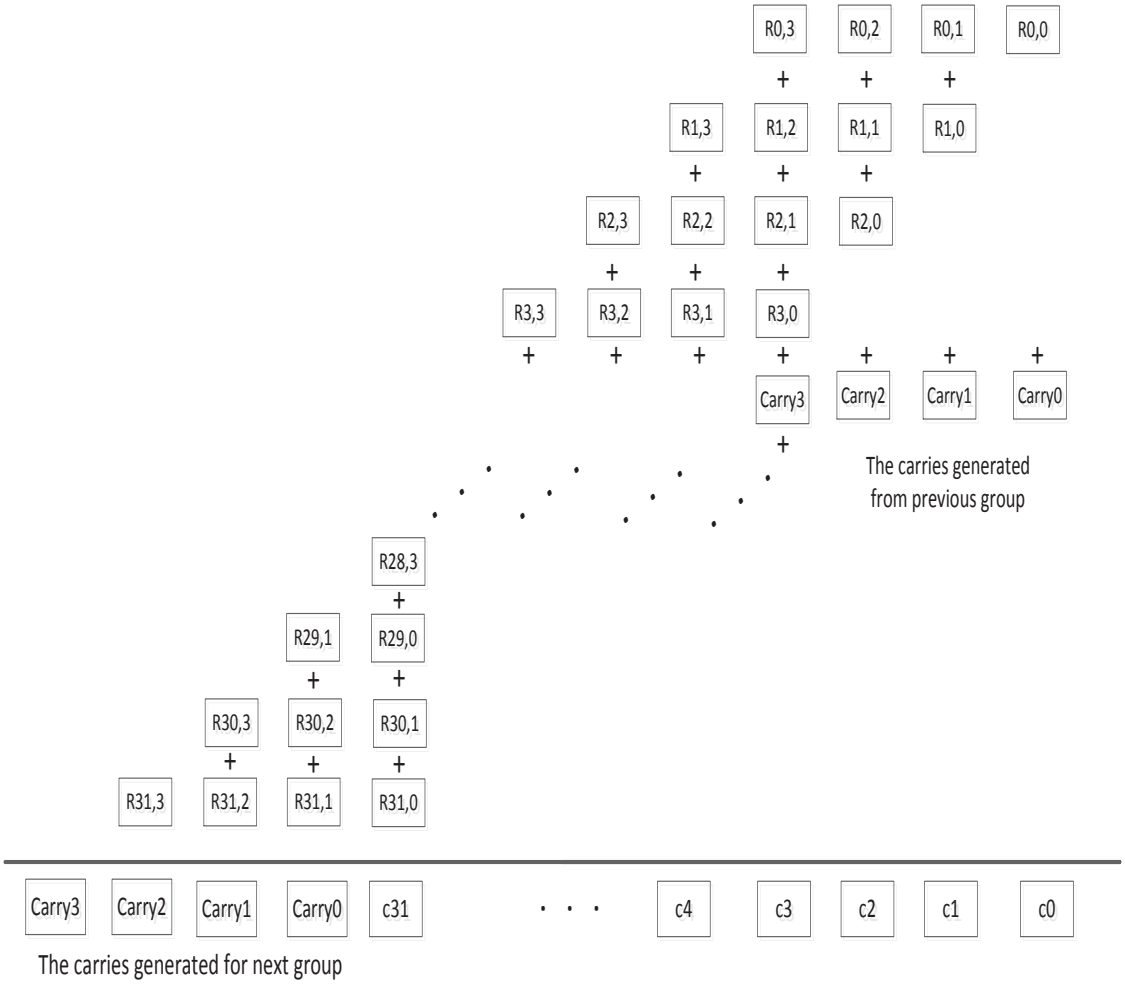

Figure 7.2: Two-stage pipeline carry resolving unit.

7.4 The Architecture for Modular Multiplication

Memory-based in-place scheme is used for the FFT design. The 1024-point FFT can be decomposed into 2 stages of 32-point FFT. At each stage, a total of 1024 samples are processed through a radix-32 butterfly unit. The radix-16 unit can be recursively used four times to complete one radix-32 butterfly computation. A group of 32 input samples are read from memory, permutated into a proper order by the Interchange Unit, fed to the radix-16 unit to process four times for one radix-32 butterfly computation, modular multiplied by the twiddle factors stored in ROMs, permutated again by the Interchange Unit, and written back to the memory. The memory needs to be partitioned into 32 banks with 32 words in each bank. An in-place memory addressing scheme can be derived to ensure there is no memory access conflict in reference to [64] [65]. The data needs to be read from and written to the memory concurrently, so dual-port SRAMs shown in Fig. 7.3 are used to store two multiplicands X and Y.

One radix-16 unit are used both for FFT and IFFT to multiply, for instance, X by Y. In the first stage of FFT or IFFT, the 8 units of 64-bit ModMuls are used to multiply the processed samples with twiddle factors. In the second stage of FFT, the same 8 units of 64-bits ModMuls can be reused to multiply $FFT(X)$ and $FFT(Y)$ for component-wise product to calculate the product of X and Y, or multiply $FFT(X)$ and $FFT(X)$ to obtain X^2. After the IFFT, a group of 32 data are fed into the resolve carries unit.

The FFT forms of M' and M are precomputed and stored in the single-port SRAMs to reduce the computation complexity. The large-number addition unit shown in Fig. 7.3 uses the same hierarchical carry-look-ahead scheme as in resolve carries

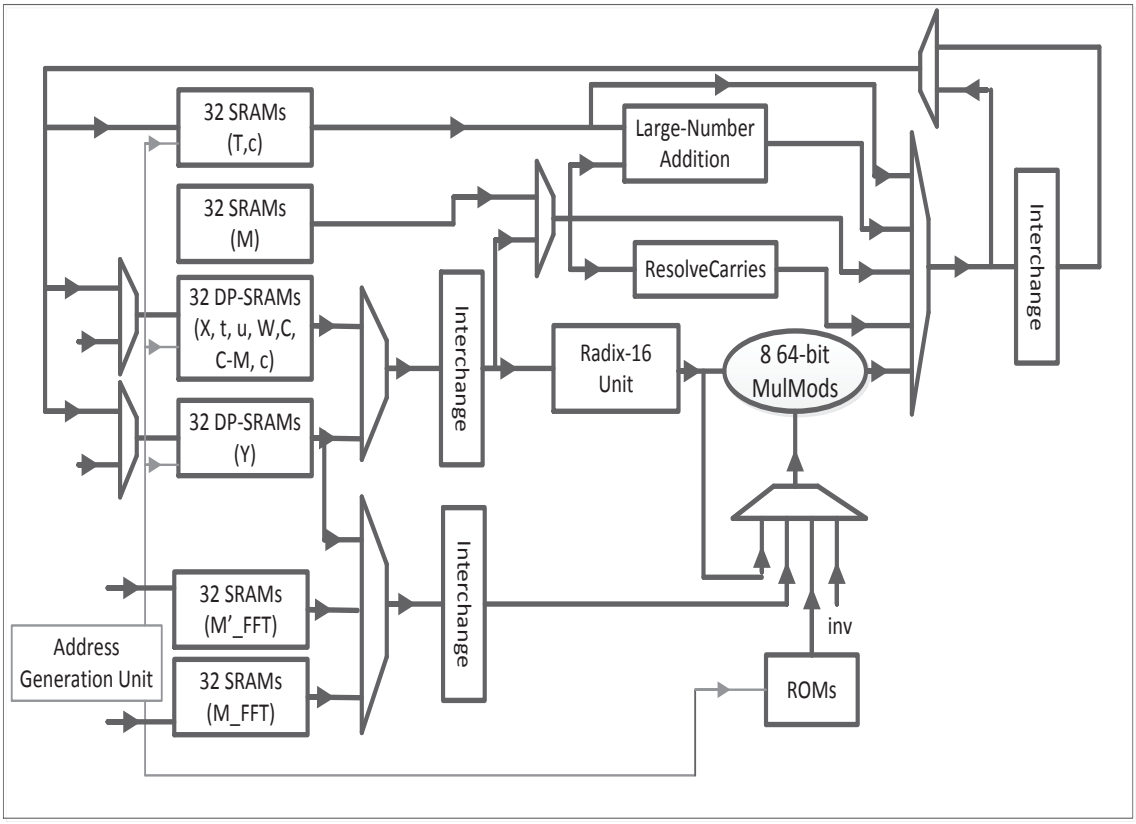

Figure 7.3: The Architecture for Modular Multiplication

unit. The large number addition performs the operation of *Step* 6 in Algorithm 7.1. The comparison in *Step*8 is actually a large-number subtraction. The 2's complement of M is precomputed and stored in the SRAMs so the large-number addition unit in Fig. 7.3 is reused for the subtraction.

Algorithm 7.2 Modular Exponentiation Using FFT Multiplication

Procedure Exponentiation(P,E,C, M): $C = P^E (\mathrm{mod}\, M)$
Inputs: P = plain text; E = exponent = $[e_{k-1}e_{k-2}...e_0]$, $e_i \in [0,1]$;M = module of m bits.
Precomputation: $n = \left\lceil log_2^M \right\rceil$, $R = 2^n$,$R' = R^{-1} \mathrm{mod}\, M$,$P' = P \times R \,\mathrm{mod}\, \mathrm{M}$, $cur = 1 \times R \,\mathrm{mod}\, \mathrm{M}$
1. for i in $k-1$ to 0 do
2. $cur \leftarrow IFFT(\, FFT(cur) \,\odot\, FFT(cur)\,) \,\mathrm{mod}\, M$;
3. if $e_i = 1$ then
4. $cur \leftarrow IFFT(\, FFT(cur) \,\odot\, FFT(P')\,) \,\mathrm{mod}\, M$;
5. end for;
Postcomputation: $C \leftarrow cur \times R' \mathrm{mod}\, M$;
end procedure.

7.5 Modular Exponentiation Using Strassen Multiplication

We use the algorithm shown in Algorithm 7.2, similar to the MSB-first algorithm in [71], for modular exponentiation. In this algorithm, P is a k-bit message with a value less than the modulus M and denote E as a m-bit exponent or key. The multiplication is similar to one cyclic convolution, which is a component-wise production in the FFT domain. By taking this advantage, the square operation in Step 2 of Algorithm 7.2 can be achieved by performing a component-wise production of the two same FFT results. In this way, we only need one FFT operation and one IFFT operation instead of two FFT operations and one IFFT operation for the multiplication in Step 2. The FFT results of P' only need to be calculated once and stored in SRAMs so the multiplication in Step 4 only needs one FFT operation and IFFT operation similar to Step 2. In all, by taking advantage of the FFT multiplication, we can manage to reduce one third of the calculation time for one large-number multiplication in Step 2 and 4. In the hardware implementation, the m-bit exponent is stored in registers

and can be fed into a state machine, leaving the state machine to take care of the modular exponentiation algorithm.

7.6 Hardware Implementation and Performance Comparisons

The design of the large-number multiplier was implemented using System Verilog. In order to compare with [37],which is the first design target for 8,192 RSA to the best of our knowledge, the design is synthesized using Altera Quartus-II synthesize tool. After place and route, the design is implemented on Altera's Stratix-V 5SGSMD8N1F45I2 FPGA. The resource utilized by the modular multiplication are listed in Table 1.

Table 7.1: TABLE 1. Synthesis result and comparison

Logic Utilization	Our Design	8192-bit RSA [37]
Combinational ALU	213,677	32,262
Dedicated Logic Register	89,007	82,023
DSP Blocks	72	—-
Block Memory bits	483,328	—-
Cycles per One MulMod	2330	32,776
Cycles per One Fast MulMod	2030	32,776

The design can also support 8,192 or 12,288-bit RSA encryption if the base is set to 16 bits or 24 bits. The FPGA Operation Maximum Frequency (OMF) of the modular multiplier is 209 MHz. It takes 2330 cycles to calculate one modular multiplication with two FFTs and one IFFTs for one multiplication. In our RSA using FFT multiplication, we only need one FFT and IFFT for one multiplication

and it takes 2030 cycles for one fast modular multiplication as shown in Table 2. It takes 9.7 μs to complete one modular multiplication when the design operates at 209 MHz. The proposed modular multiplication is about 16.1 times faster than the RSA co-processor running at the same frequency reported in [37].

The design in [37] uses two modular multipliers in parallel to perform the modular exponentiation. In our design, one modular multiplier is recursively used. Our design takes 0.159 s to complete one modular exponentiation while the design in [37] takes 1.28 s, when both running at 209 MHz. The proposed modular multiplication is about 8 times faster than the RSA co-processor in [37].

In order to be referenced and compared by future designs, the multiplier ASIC was also synthesized for 90nm technology, using the Synopsys Design Compiler, the DesignWare building block libraries, and IBM 90nm CMOS 9FLP standard-cell library. The SRAMs in the design come from Synopsys Designware library. Table 2 lists the synthesis results for the RSA chips. The number of logic equivalent gates (two-input NAND) of the chip is 5,300K gates.

Table 7.2: TABLE 2. Synthesis results using 90-nm CMOS technology (IBM 90nm 9FLP process)

Logic Utilization	RSA Chip
Core Area	11.7 mm^2
Clock Frequency (MHz)	320 MHz
Clock Voltage	1.32 V

The designs for RSA in [4-11] are targeted for 1,024 or 2,048-bit RSA applications. There are no reports about performance of 8K-bit RSA application in these design. Usually different designs have their ASIC implementation results with key size 1,024 bits to compare with others. In order to fairly compare with these designs, we establish

Table 7.3: TABLE 3. Modular Multiplication and Exponentiation Time (Operating at 320 MHz in ASIC)

#Bits	Multiplication Time	Exponentiation Time	Throughput
8,192	6.34 μs	0.104 s (worst)	78.8K bits/s
12,288	6.34 μs	0.156 s (worst)	78.8K bits/s

Table 7.4: TABLE 4. Implementation Comparisons

Ref	Technology	Area (gates)	Period(ns)	MulMod/s	Key Size	$BitMul/(gates \cdot s)$	$BitMul/(gates \cdot s \cdot freq)$
[69]	0.5 μm CMOS	156K gt	20.0	94.2K	1,024	2.53 M	0.051
[70]	0.18 μm CMOS	148K gt	2.2	438.6K	1,024	12.4 M	0.027
[28]	0.13 μm CMOS	139K gt	2.0	648.6K	1,024	19.6 M	0.039
Ours	90 nm CMOS	5,300K gt	3.1	157.6	12,288	18.0 M	0.056

the concept that compares the bit multiplications that one gate of hardware can complete in one seconds. Although the designs in [28] [69] and [70] use different optimization strategies to improve the interleaved Montgomery algorithm, their goal are all same, which is to complete the original interleaved Montgomery algorithm. So we use the original interleaved Montgomery algorithm as the standard to estimate its bit multiplications. The original interleaved Montgomery algorithm has $2M^2 + M$ bit multiplication with M the bit size so we have (7.4). Table 4 lists the implementation comparison. From that table we can see, the proposed design ranked No. 2 when we compared with the $BitMulsPerGatePerSec$. If all the designs operating at the same frequency, our design could beat all the rest of designs.

$$BitMulsPerGatePerSec = \frac{2M^2 + M}{Gates \cdot Seconds} \tag{7.4}$$

To understand the arithmetic cost of the interleaved Montgomery algorithm and FFT based algorithm, we implement two different modular multiplication algorithms in carefully tuned MIPS64 assembly and count the number of ALU operations for each

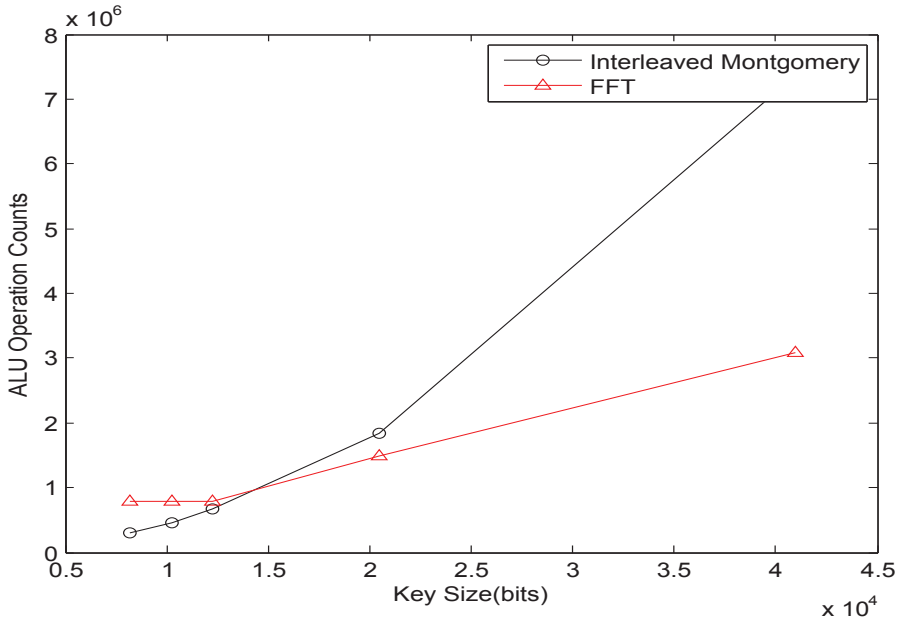

Figure 7.4: Operation Counts of Two Different Algorithms

as shown in Fig. 4. Although the interleaved Montgomery algorithm is more efficient than the FFT based algorithm for 8K and 12K RSA, our well-designed hardware can still beats the other designs. When the key size is great than 20K bits, the FFT based algorithm is more efficient and should be the top option for RSA hardware design.

7.7 Conclusions

In this research, a novel and fast modular multiplication and exponentiation architecture is presented for RSA with large key sizes. Instead of using the well-known interleaved version of Montgomery multiplication, we combined the Strassen multiplication and Montgomery reduction for the modular multiplier design. The design support both 8K- and 12K-bit modular multiplication and exponentiation. To the best of our knowledge, it is the first design that can support 12K-bit modular multi-

plication and exponentiation for RSA. The design can complete one 8K- and 12K-bit RSA operation in 0.104 s and 0.156 s operating at 320 MHz, which is the fastest design to the best of our knowledge.

Chapter 8

Conclusions

8.1 Summary of Results

This dissertation is devoted to using the GPU and custom hardware to accelerate the existing fully homomorphic encryption schemes, and introducing the new hardware design method for RSA cryptosystem based on FFT multiplication.

- Firstly, we present the first GPU implementation of a fully homomorphic encryption scheme. We develop efficient techniques for large integer arithmetic operations to support the higher level primitives of the Gentry-Halevi FHE. We combined Emmart and Weems' implementation of Strassen's FFT multiplication with Barrett reduction for a high-speed modular multiplication on a GPU. In addition, we tailor the encryption and recryption functions to make optimal use of GPU features as well as to avoid obstacles, such as lack of support for recursive operations. We also develop a pre-computation strategy to further enhance the efficiency of the encryption primitive. We gained about 8 times speedup when running our implementation on a server equipped with a

NVIDIA Tesla C2050 GPU, compared with the CPU reference implementation in [18]. This work shows the performance of FHEs can be greatly improved by carefully choosing the target platform and by tailoring the algorithms.

- Secondly, we design a power and area efficient, high-speed large-number multiplier for Gentry-Halevi FHE scheme. The large-number multiplier is using Strassen's FFT-based multiplication algorithm. The memory-based, in-place FFT architecture was used for the FFT processor to reduce the memory usage. We use a number of design optimization strategies to improve the performance and reduce the area of the Radix-16 unit. The multiplier was synthesized for 90nm technology with an estimated core area 45.3 mm^2. Experimental results show that the proposed multiplier is about 2 times faster than GPU and 29 times faster than CPU, and its power consumption is less than 1 watt. To the best of our knowledge, this is the fastest multiplier that has been implemented using VLSI design for fully homomorphic encryption.

- Thirdly, we follow our previous step to use GPU to accelerate the crucial part in the leveled FHE scheme. The CRT method is used for the efficient large-number matrix-vector multiplication, gained 7.8 speedup compared to the NTL library function. The GPU is used to accelerate the vector operation process, accounting for 99.6% of the total computation, to further accelerate the matrix-vector multiplication. In the GPU implementation, we manage to overlap the calculation process and data transfer process to improve the computation efficiency. Experimental results show the proposed CRT-based method with GPU implementation gains about 273.6 times speedup when compared with the NTL library function and 35.2 times speedup when compared with the same CRT-

based method on CPU.

- Finally, we present a novel and fast modular multiplication and exponentiation architecture for large key-size RSA cryptosystem. We paired the FFT multiplication algorithm with Montgomery reduction for the modular multiplication design instead of using the traditional interleaved version of Montgomery multiplication in the RSA hardware design. The proposed design can support both 8K- and 12K-bit modular multiplication and exponentiation. To the best of our knowledge, it is the first design that can support 12K-bit modular multiplication and exponentiation for RSA. The design can complete one 8K- and 12K-bit RSA operation in 0.104 s and 0.156 s operating at 320 MHz, which is the fastest design to the best of our knowledge.

8.2 Overview of Contribution

In this dissertation, we present the first GPU acceleration and the hardware design of a large-number multiplier based on FFT multiplication both for Gentry-Halevi's FHE implementation. We follow this path and continue to use GPU to accelerate the BGV leveled FHE scheme, which is more efficient than the Gentry-Halevi's FHE implementation. Since we designed a very efficient hardware multiplier, we bring the FFT multiplication for the hardware design of RSA cryptosystems by combining the FFT multiplication and Montgomery reduction instead of the traditional interleaved Montgomery multiplication.

In this dissertation, we are tackling the existing FHE schemes instead of proposing new FHE schemes. We are trying to using the more efficient computation algorithms such as FFT based multiplication and Chinese Remainder Theorem to accelerate

the basic primitives such as modular multiplication in this existing FHE schemes. For the GPU acceleration, the targeted platform is the NVIDIA's general-purpose GPU. The GPU implementations need some requirements for the memory spaces and architecture support of the GPU for instance the Fermi architecture. With slightly changes, the GPU implementations can be migrated to the other NVIDIA GPU platforms.

8.3 Recommendations for Future Work

Although Gentry's original construction is inefficient and impractical, recent new constructions have significantly improved the efficiency of fully homomorphic encryption. Especially, the leveled fully homomorphic encryption proposed by Brakerski, Gentry and Vaikuntanathan outstands itself, with a asymptotically better FHE system. In our research, we only use GPU to accelerate the most computation-intensive part in the encryption of the leveled fully homomorphic encryption scheme. Future work can use GPU to accelerate the whole leveled FHE scheme including encryption, decryption, refresh process, homomorphically addition and multiplication. Because of the leveled FHE scheme is much more efficient than the Gentry-Halevi FHE scheme. We expect to see a more promising GPU acceleration results for real life deployment. If the GPU can get a very good acceleration results, it means the custom hardware can attain a similar or even better acceleration results. Compared with Gentry-Halevi FHE scheme, the leveled FHE scheme uses 512, 1024 or 2,048-bit large-number multiplications, which are also widely used in RSA cryptosystems. A number of efficient modular multipliers using interleaved Montgomery multiplication are used for the RSA hardware design, which can be useful for the leveled FHE hardware design.

Bibliography

[1] J. D. Cohen and M. J. Fischer, "A robust and verifiable cryptographically secure election scheme," in *FOCS*, vol. 85, 1985, pp. 372–382.

[2] E. Kushilevitz and R. Ostrovsky, "Replication is not needed: Single database, computationally-private information retrieval," in *Foundations of Computer Science, 1997. Proceedings., 38th Annual Symposium on*. IEEE, 1997, pp. 364–373.

[3] M. Naehrig, K. Lauter, and V. Vaikuntanathan, "Can homomorphic encryption be practical?" in *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*. ACM, 2011, pp. 113–124.

[4] R. L. Rivest, L. Adleman, and M. L. Dertouzos, "On data banks and privacy homomorphisms," *Foundations of secure computation*, vol. 32, no. 4, pp. 169–178, 1978.

[5] S. Goldwasser and S. Micali, "Probabilistic encryption," *Journal of computer and system sciences*, vol. 28, no. 2, pp. 270–299, 1984.

[6] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[7] T. ElGamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," *Information Theory, IEEE Transactions on*, vol. 31, no. 4, pp. 469–472, 1985.

[8] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *EUROCRYPT 99*. Springer, 1999, pp. 223–238.

[9] I. Damgård and M. Jurik, "A generalisation, a simplication and some applications of Paillier's probabilistic public-key system," in *Public Key Cryptography*. Springer, 2001, pp. 119–136.

[10] M. Ajtai and C. Dwork, "A public-key cryptosystem with worst-case/average-case equivalence," in *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*. ACM, 1997, pp. 284–293.

[11] O. Regev, "New lattice-based cryptographic constructions," *Journal of the ACM (JACM)*, vol. 51, no. 6, pp. 899–942, 2004.

[12] J. Benaloh, "Dense probabilistic encryption," in *Proceedings of the workshop on selected areas of cryptography*, 1994, pp. 120–128.

[13] D. Naccache and J. Stern, "A new public key cryptosystem based on higher residues," in *Proceedings of the 5th ACM conference on Computer and communications security*. ACM, 1998, pp. 59–66.

[14] D. Boneh, E.-J. Goh, and K. Nissim, "Evaluating 2-DNF formulas on ciphertexts," in *Theory of cryptography*. Springer, 2005, pp. 325–341.

[15] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proc. of the 41st Annual ACM Symposium on Theory of Computing*, 2009, pp. 169–178.

[16] M. Van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan, "Fully homomorphic encryption over the integers," in *Advances in Cryptology–EUROCRYPT 2010*. Springer, 2010, pp. 24–43.

[17] N. P. Smart and F. Vercauteren, "Fully homomorphic encryption with relatively small key and ciphertext sizes," in *Public Key Cryptography–PKC 2010*. Springer, 2010, pp. 420–443.

[18] C. Gentry and S. Halevi, "Implementing Gentry's fully-homomorphic encryption scheme," *Advances in Cryptology–EUROCRYPT 2011*, pp. 129–148, 2011.

[19] Z. Brakerski and V. Vaikuntanathan, "Fully homomorphic encryption from ring-lwe and security for key dependent messages," in *Advances in Cryptology–CRYPTO 2011*. Springer, 2011, pp. 505–524.

[20] N. P. Smart and F. Vercauteren, "Fully homomorphic SIMD operations," *Designs, Codes and Cryptography*, pp. 1–25, 2011.

[21] Z. Brakerski and V. Vaikuntanathan, "Efficient fully homomorphic encryption from (standard) lwe," in *Foundations of Computer Science (FOCS), 2011 IEEE 52nd Annual Symposium on*. IEEE, 2011, pp. 97–106.

[22] C. Gentry, S. Halevi, and N. P. Smart, "Fully homomorphic encryption with polylog overhead," in *Advances in Cryptology–EUROCRYPT 2012*. Springer, 2012, pp. 465–482.

[23] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(Leveled) fully homomorphic encryption without bootstrapping," in *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*. ACM, 2012, pp. 309–325.

[24] V. Vaikuntanathan, "Computing blindfolded: New developments in fully homomorphic encryption," in *Foundations of Computer Science (FOCS), 2011 IEEE 52nd Annual Symposium on*. IEEE, 2011, pp. 5–16.

[25] S. A. Manavski, "Cuda compatible GPU as an efficient hardware accelerator for AES cryptography," in *Signal Processing and Communications, 2007. ICSPC 2007. IEEE International Conference on*. IEEE, 2007, pp. 65–68.

[26] A. Moss, D. Page, and N. P. Smart, "Toward acceleration of RSA using 3D graphics hardware," in *Cryptography and Coding*. Springer, 2007, pp. 364–383.

[27] X. Zhang and K. Parhi, "High-speed VLSI architectures for the AES algorithm," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 12, pp. 957–967, 2004.

[28] M.-D. Shieh, J.-H. Chen, H.-H. Wu, and W.-C. Lin, "A new modular exponentiation architecture for efficient design of RSA cryptosystem," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 9, pp. 1151–1161, 2008.

[29] W. Wang, Y. Hu, L. Chen, X. Huang, and B. Sunar, "Accelerating fully homomorphic encryption using GPU," in *Proc. of 2012 IEEE Conference on High Performance Extreme Computing*. IEEE, 2012, pp. 1–5.

[30] ——, "Exploring the Feasibility of Fully Homomorphic Encryption," *IEEE Transactions on Computers*, Aug. 2013.

[31] W. Wang, X. Huang, N. Emmart, and C. Weems, "VLSI design of a large number multiplier for fully homomorphic encryption," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Nov. 2013.

[32] A. López-Alt, E. Tromer, and V. Vaikuntanathan, "On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption," in *Proceedings of the 44th symposium on Theory of Computing*. ACM, 2012, pp. 1219–1234.

[33] *CUDA C PROGRAMMING GUIDE*, 5th ed., NVIDIA, July 2013.

[34] P. Montgomery, "Modular Multiplication without Trial Division," *Mathematics of Computation*, vol. 44, no. 170, pp. 519–521, 1985.

[35] W. Wang and X. Huang, "A novel fast modular multiplier architecture for 8,192-bit RSA cryposystem," in *Proc. of 2013 IEEE Conference on High Performance Extreme Computing*. IEEE, 2013.

[36] P. Barrett, "Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor," in *Advances in cryptology (CRYPTO 86)*. Springer, 1987, pp. 311–323.

[37] C. P. Rentería-Mejía, V. Trujillo-Olaya, and J. Velasco-Medina, "Design of an 8192-bit RSA cryptoprocessor based on systolic architecture," in *2012 VIII Southern Conference on Programmable Logic (SPL)*. IEEE, 2012, pp. 1–6.

[38] A. Karatsuba and Y. Ofman, "Multiplication of many-digital numbers by automatic computers," in *Proc. of the USSR Academy of Sciences*, no. 2, 1962, pp. 293–294.

[39] D. Knuth, *The Art of Computer Programming*. Addison-Wesley, 2006, vol. 2.

[40] A. Schönhage and V. Strassen, "Schnelle Multiplikation Grosser Zahlen," *Computing*, vol. 7, no. 3, pp. 281–292, 1971.

[41] N. Emmart and C. C. Weems, "High precision integer multiplication with a GPU using Strassen's algorithm with multiple FFT sizes," *Parallel Processing Letters*, vol. 21, no. 03, pp. 359–375, 2011.

[42] J. Solinas, "Generalized Mersenne Numbers," *Technical Reports*, 1999.

[43] J. W. Cooley and J. W. Tukey, "An Algorithm for the Machine Calculation of Complex Fourier Series," *Mathematics of Computation*, vol. 19, no. 90, pp. 297–301, 1965.

[44] G. D. Sutter, J.-P. Deschamps, and J. L. Imaña, "Modular multiplication and exponentiation architectures for fast RSA cryptosystem based on digit serial computation," *IEEE Transactions on Industrial Electronics*, vol. 58, no. 7, pp. 3101–3109, 2011.

[45] X. Cui, Y. Chen, and H. Mei, "Improving performance of matrix multiplication and FFT on GPU," in *Proc. of 15th International Conference on Parallel and Distributed Systems (ICPADS 2009)*. IEEE, 2009, pp. 42–48.

[46] N. Govindaraju, N. Raghuvanshi, and D. Manocha, "Fast and approximate stream mining of quantiles and frequencies using graphics processors," in *Proc. of the 2005 ACM SIGMOD international conference on Management of data*. ACM, 2005, pp. 611–622.

[47] J. Baladron Pezoa, D. Fasoli, and O. Faugeras, "Three applications of GPU computing in neuroscience," *Computing in Science and Engineering*, no. 99, pp. 1–1, 2011.

[48] C. Mclvor, M. McLoone, and J. McCanny, "Fast Montgomery modular multiplication and RSA cryptographic processor architectures," in *Conference Record of the Thirty-Seventh Asilomar Conference on Signals, Systems and Computers, 2003.*, vol. 1. IEEE, 2003, pp. 379–384.

[49] A. Daly and W. Marnane, "Efficient architectures for implementing Montgomery modular multiplication and RSA modular exponentiation on reconfigurable logic," in *Proceedings of the 2002 ACM/SIGDA Tenth International Symposium on Field-Programmable Gate Arrays*. ACM, 2002, pp. 40–49.

[50] P. Giorgi, T. Izard, A. Tisserand *et al.*, "Comparison of modular arithmetic algorithms on GPUs," 2009.

[51] D. Bailey, "FFTs in external of hierarchical memory," in *Proceedings of the 1989 ACM/IEEE conference on Supercomputing*. ACM, 1989, pp. 234–242.

[52] V. Shoup, "NTL: A Library for Doing Number Theory," 2001.

[53] *The GNU Multiple Precision Arithmetic Library.*, 2010, http://gmplib.org/, Version 5.0.1.

[54] A. Cohen and K. Parhi, "GPU accelerated elliptic curve cryptography in GF(2m)," in *IEEE International Midwest Symposium on Circuits and Systems (MWSCAS)*, 2010.

[55] D. Cousins, K. Rohloff, C. Peikert, and R. Schantz, "SIPHER: Scalable implementation of primitives for homomorphic encryption–FPGA implementation using simulink," *High Performance Extreme Computing Conference*, 2011.

[56] Y. Doröz, E. Oztürk, and B. Sunar, "Accelerating fully homomorphic encryption in hardware."

[57] C. Burnikel, R. Fleischer, K. Mehlhorn, and S. Schirra, "Efficient exact geometric computation made easy," in *Proc. of the Fifteenth Annual Symposium on Computational Geometry*. ACM, 1999, pp. 341–350.

[58] C. K. Yap and V. Sharma, *Robust Geometric Computation*. Springer, 2008.

[59] V. Karamcheti, C. Li, I. Pechtchanski, and C. Yap, "A Core Library for Robust Numeric and Geometric Computation," in *Proc. of the Fifteenth Annual Symposium on Computational Geometry*. ACM, 1999, pp. 351–359.

[60] S. Yazaki and K. Abe, "VLSI Design of Karatsuba Integer Multipliers and Its Evaluation," *IEEE Trans. on Electronics, Information and Systems*, vol. 128, pp. 220–230, 2008.

[61] ——, "An Optimum Design of FFT Multi-Digit Multiplier and Its VLSI Implementation," *Bulletin of the University of Electro-Communications*, vol. 18, no. 1, pp. 39–45, 2006.

[62] K. Kalach and J. P. David, "Hardware Implementation of Large Number Multiplication by FFT with Modular Arithmetic," in *Proc. of the 3rd Internationa IEEE-NEWCAS Conference*. IEEE, 2005, pp. 267–270.

[63] L. Jia, Y. Gao, and H. Tenhunen, "A pipelined shared-memory architecture for FFT processors," in *Proc. of 42nd IEEE Midwest Symposium on Circuits and Systems*, vol. 2. IEEE, 1999, pp. 804–807.

[64] L. Johnson, "Conflict free memory addressing for dedicated FFT hardware," *IEEE Trans. on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 39, no. 5, pp. 312–316, 1992.

[65] H. Lo, M. Shieh, and C. Wu, "Design of an Efficient FFT Processor for DAB System," in *Proc. IEEE Int. Symp. Circuits Systems*, vol. 4. IEEE, 2001, pp. 654–657.

[66] N. Emmart and C. Weems, "High precision integer addition, subtraction and multiplication with a graphics processing unit," *Parallel Processing Letters*, vol. 20, no. 4, pp. 293–306, 2010.

[67] C. Gentry, S. Halevi, and N. P. Smart, "Homomorphic evaluation of the AES circuit," in *Advances in Cryptology–CRYPTO 2012*. Springer, 2012, pp. 850–867.

[68] J. Grossschadl, "The chinese remainder theorem and its application in a high-speed RSA crypto chip," in *Computer Security Applications, 2000. ACSAC'00. 16th Annual Conference*. IEEE, 2000, pp. 384–393.

[69] T.-W. Kwon, C.-S. You, W.-S. Heo, Y.-K. Kang, and J.-R. Choi, "Two implementation methods of a 1024-bit RSA cryptoprocessor based on modified Montgomery algorithm," in *The 2001 IEEE International Symposium on Circuits and Systems, 2001.*, vol. 4. IEEE, 2001, pp. 650–653.

[70] Q. Liu, F. Ma, D. Tong, and X. Cheng, "A regular parallel RSA processor," in *Circuits and Systems, 2004. MWSCAS'04. The 2004 47th Midwest Symposium on*, vol. 3. IEEE, 2004, pp. iii–467.

[71] G. D. Sutter, J.-P. Deschamps, and J. L. Imaña, "Modular multiplication and exponentiation architectures for fast RSA cryptosystem based on digit serial computation," *IEEE Transactions on Industrial Electronics*, vol. 58, no. 7, pp. 3101–3109, 2011.